XFaceMaker/UIMS

Version 4.0/Motif2.1

Reference Manual

Nova Software Labs

Published by:

Nova Software Labs
8, rue de Hanovre
75002 Paris, France
Tel: +33 1 58 18 61 61
Fax: +33 1 58 18 61 60
email: sales@NovaSoftwareLabs.com or sales@nsl.fr.
Web: http://www.NovaSoftwareLabs.com or http://www.nsl.fr.

Document Number: XFMUIMSRefMan v. 7.1.1.
Document date: November 29, 2002

Acknowledgments

The following people have contributed to the development of XFaceMaker:

Alain Bastard, Genevieve Beaujard, Nicole Brion, Alan Char, Eric Durocher, Marc Durocher, France Geze, Eric Herbaut, Elizabeth Huffer, Ion Filotti, Jérôme Maloberti, Sara Sautin, Eric Touchard, Philippe Volle.

Many thanks to all of them and to several others.

Table of Contents

List of Tables

List of Figures

CHAPTER 1: Introduction

Welcome to XFaceMaker. This Reference Manual describes the functions and most of the features of XFaceMaker in a reference format. More detailed explanations on how to use these features can be found in the *XFaceMaker User's Guide.*

In addition, XFaceMaker includes extensive on-line help, including help on the FACE functions. For detailed information on the Motif, Xt, and C functions callable in XFaceMaker, consult the appropriate documentation, or the on-line man pages that are provided with the X Window System.

The XFaceMaker distribution includes a number of examples that illustrate how to do things. Please refer to the **demos** directory in the installed tree.

Before You Start

This manual assumes that you are familiar with the C programming language and, to a lesser extent, with the Unix operating system and the X Window system.

We assume that you have access to a reference manual for the Motif widgets. The Motif Style Guide can help you with ideas on what you should be trying to achieve with your interface. Some of the books which describe the principles involved in both the X Window System and the Motif widget set are listed in the bibliography.

You should also know how to use a window manager and xterm or a similar terminal emulator, and you should be able to open, resize, iconify, and close windows.

1.1 What You Need to Run XFaceMaker

To run any application using the X Window System you need:

- A Unix host computer. The Unix host will run all the applications. The application makes calls to the *Xlib* library for displaying, and for processing keyboard and mouse events. The host must be set up to communicate with other machines on a network (TCP/IP). The host system must include development software: compiler, linker, etc.
- A window manager. Although XFaceMaker can run with different window managers, it is guaranteed to work with the OSF/Motif window manager mwm.
- A graphic display consisting of a monitor, a keyboard and a mouse.

There are two main types of graphic displays on which you can run the X Window System (and thus XFaceMaker):

- *Unix workstations*. These are Unix computers with a graphic screen. They run the X server as a Unix process.
- *X terminals*. X terminals run only the X server as a stand-alone program. The application itself runs on the remote Unix host.

Motif based applications require a screen resolution of at least 1024 x 768 pixels. If you plan to use color, make sure you have 256 colors. It is unreasonable to attempt to create a Motif-based application with only 16 colors.

The following libraries must be installed on the host computer:

- *The Xlib*. Host applications call the Xlib library when they want to use the X server. The Xlib is usually installed as /usr/lib/libX11.a.
- *The X Intrinsics library*. Called the Xt library, it is usually installed as /usr/lib/libXt.a.
- *The Motif 2.1 widget set*. It is usually installed as /usr/lib/libXm.a.
  This version of XFaceMaker is designed to work with Motif 2.1 or Open Motif 2.1.
- *The* XFaceMaker *libraries:*

  libFm.a -- The interpreting library for using interpreted scripts.
  libFm_c.a -- The C code library for fully compiled applications.
  libFm_e.a -- The XFM executable library for applications which include XFaceMaker itself, or to rebuild XFM.

The following include files must be installed on the host computer:

- The Xlib and X Intrinsics include files. These are usually installed in /usr/include/X11.
- The Motif include files. These are usually installed in /usr/include/Xm.
- The XFaceMaker include file, Fm.h. This is usually installed in /usr/include.

1.2 Keyboard Conventions

Key combinations are expressed as follows:

Format	Example	Meaning
Key1--Key2	Ctrl--A	Hold down the Control key as you press, then release key A. The shift key is not used.
Key1 Key2	Esc a	Press and release the Escape key, then press and release key a.
	Esc A	Press and release the Escape key, then press and release key a while pressing the shift key.

Table 1.1: Keyboard Conventions

Notice how in the case of the Esc modifier the two keys are pressed in succession, whereas in the case of the Ctrl modifier the key is pressed *while* the Ctrl key is held down.

1.3 Typesetting Conventions

The following typesetting conventions are used in this manual:

Type style	Used for
`typewriter`	Anything you should type as it appears, code, and key combinations.
italic	Mouse buttons and emphasis.
bold slant	Menu items and editing commands.

Table 1.2: Typesetting Conventions

1.4 Using the Mouse

We assume you are using a three button mouse. If not, you need to know which combinations of keyboard and mouse actions correspond to *Select, Menu,* and *Custom*.

Mouse buttons

Mouse buttons respond differently in Build mode and in Try mode. In Build mode the mouse buttons are used as follows:

Button1: **Select**--Used to select/deselect widgets on the desktop or in the Widget List or Tree, to select menu items, open a menu, or drag.

Button2: **Menu**---Used to select widgets on the desktop, and popup the Resources or the Resource Editor boxes.

Button3: **Custom--**Used to select the *parent* of the *current* widget.

In Try mode, most button actions are determined by the widget resources and callbacks defined for the particular interface. By default, PopupMenus are popped up using the Custom button.

Mouse actions

This manual uses the following terms for mouse actions:

Press Hold down a mouse button.

Click and double-click Quickly press and release once (click) or twice (double-click) the **Select** mouse button without moving the mouse. Clicking selects. Double-clicking is for editing actions.

Shift-click Hold down the Shift key, and click the left mouse button.

Drag Move the mouse while pressing the **Select** mouse button.

Rubber band A rectangular outline displayed when the cursor is dragged. One corner is fixed at the point where the button was pressed and the opposite corner follows the pointer. Releasing the button creates a rectangle that defines either the size of a widget or a selection area.

Outline box This is the outline of a widget displayed as the widget is moved so that it can be placed correctly.

1.5 Organization of This Manual

This manual is organized as follows:

1. **Introduction** provides an overall view of this Reference Manual.
2. **NSL Product Support** explains how to contact us.
3. **Architecture** describes the dual-process, and error protection.
4. **Command Window** describes the features of the Command window.
5. **Editing Boxes** lists the editing boxes available in XFaceMaker in alphabetical order, and describes the features of each box.
6. **The FACE Language** is a brief, but comprehensive presentation of the FACE language.
7. **FACE in XFaceMaker** describes the use of the FACE language specifically in the XFaceMaker context.
8. **Memory Management** describes how XFaceMaker handles memory management.
9. **Structures** describes those registered or defined in XFaceMaker.
10. **The Genappli Utility** describes the tool used to generate application files associated to an interface.
11. **New Widget Classes** describes the resources and methods available for creating new widget classes in XFaceMaker.
12. **Functions** is an alphabetical list of FACE functions. Tables listing all the functions are located at the beginning of the chapter. One table lists the functions that can be called only from the application, the other lists functions that can be called from FACE and/or the application.
13. **Command Line Options** lists command line options.
14. **Environment Variables** is a list of those available in XFaceMaker.
15. **Fm File Structure** describes the structure of an interface file in the .fm format.
16. **Adding Help Files** explains how you can add new Help files or modify existing ones.
17. **Porting the C-Code Library** explains how to port the library libFm_c to a new machine.

Appendix A: **List of Functions** is a complete list of functions attached by XFaceMaker for use in FACE.

CHAPTER 2: NSL Product Support

NSL provides technical support in English and French through our office in France. The standard purchase agreement between purchasers and NSL states that NSL support includes telephone support within *reasonable* limits. NSL reserves the right to decide what amount of support is reasonable.

Please note that this support is for NSL products only and not for general X Window System information other than that relating directly to NSL products. Having said this, we wish to make it plain that the Support Department is dedicated to helping users as much as possible and allowing them to benefit from using the full capabilities of NSL products.

2.1 Recommended Actions

We recommend that one person within an organization or department be appointed as the NSL Contact Person. All inquiries should be addressed to this person in the first instance so that common problems are not repeatedly directed to the NSL Technical Support Department. This will also allow NSL to disseminate future information releases more efficiently and help users more fully.

To this end we suggest that the Contact Person do *one* of the following when the product is first installed:

- Send an electronic mail message with his or her electronic mail address to support@nsl.fr.
- Fax the NSL Technical Support Department giving a fax number or direct telephone number at which he or she can be contacted.

We would like to be informed of any problems you might have encountered while installing this particular NSL product.

2.2 Before Contacting Technical Support

If you do have to telephone Technical Support please have the following information handy when you call:

- The product name and version number you are using; for example, in XFaceMaker, it is given in the *Help* menu, under “On Version". Or enter the command xfm -revision.
- The machine you are using the product on, e.g. Sun 3-80, and the name and release of the operating system, e.g. Sun OS Release 4.0.3c, and the type of display.
- If you are using an X–Terminal; the make and model.
- The amount of RAM you have.
- The version of the X server you are using, e.g. Release 6.
- The version of the X11 you are using, e.g. Release 3
- The patch level of the Xt Intrinsic library you are using, e.g. Patch level 10.
- The version of OSF/Motif you are using, e.g. Release 2.1.1.a.
- A full description of the problem, including any error messages if applicable.
- A copy of the User’s Guide so we can point out where further help may be found.

2.3 NSL Technical Support

Technical Support is available:

- Monday to Friday (except for French national holidays).
- 9.00 am 12:30 pm – 2.00 pm to 6.00 pm (MET). 2-5pm Fridays.
- From French and English speaking support staff.
- By phone (+33-1) 58 18 61 61(ask for the Technical Support Department).
- By electronic mail addressed to support@nsl.fr
- By FAX (+33-1) 58 18 61 60 addressed to Technical Support including your return FAX number, and a telephone number where you can be reached.

2.4 Further Help

NSL gives the following courses at their premises in Paris:

- XFaceMaker, XFaceMaker/WF
- The Xlib.
- The OSF/Motif Toolkit.
- Interface Design.
- Languages (C, C++, PostScript, Java, HTML)
- WEB site creating and maintaining

Please contact the training department at (+33-1) 58 18 61 61 for more details and for information on courses tought in English.

You can order extra copies of the User’s Guide or Reference Manual. Please contact the sales department at (+33-1) 58 18 61 61 for a full list of the manuals available, and prices.

CHAPTER 3: Architecture

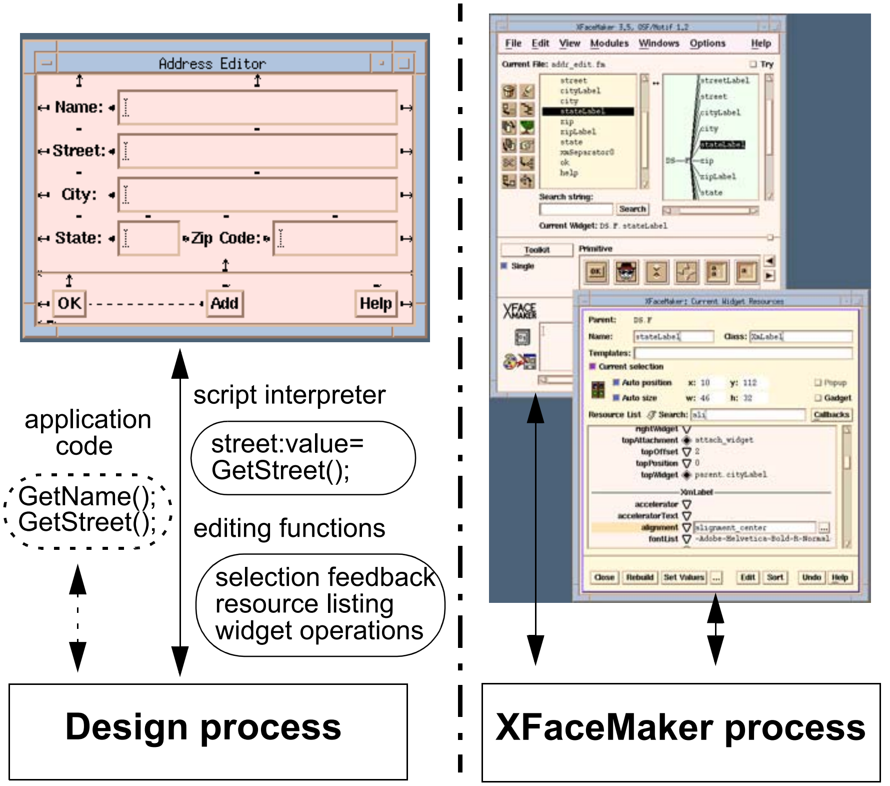
The XFaceMaker architecture is based on two distinct processes:

1. The **XFaceMaker process** is the main process which implements the XFace-Maker windows.
2. The **design process** manages the interface which is being designed by the user, and interprets the FACE scripts attached to it.

This new architecture offers two important advantages:

- Error Protection: When an error occurs in the user's interface, the design process can be completely restarted to restore a safe state.
- Dynamic Extension: Allows you to add new application functions, classes, etc. dynamically, without exiting XFaceMaker.

Dual-Process Architecture

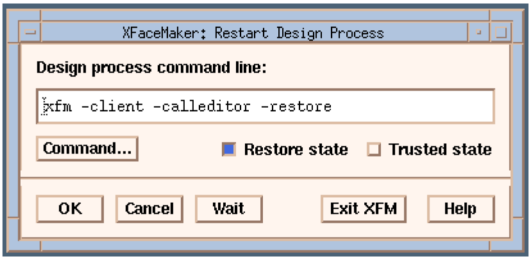


3.1 Error Protection

Error protection is a major advantage offered by the dual-process mode. In most UIMS's, as in previous versions of XFaceMaker, the interface and its behavior are implemented in the same process as the editor itself. This means that if the interface crashes (which can happen, for example, if the user calls a Motif function in a call-back with wrong arguments), then the whole program crashes. XFaceMaker solves this problem by separating the interface being designed by the user, from the XFace-Maker editor itself. In the XFaceMaker architecture, the design process can crash, but the XFaceMaker process stays alive.You can find more information on XFace-Maker's dual process architecture in the Chapter "Overview" in the *XFaceMaker User's Guide*.

3.1.1 Restarting the Design Process

The default command line displayed by the restart dialog box launches the same design process as the one which just terminated, usually xfm (or a new user-built instance of XFaceMaker), with all the command-line options specified initially, plus the -client option and the -restore option explained below.


XFaceMaker: Restart Design Process
Design process command line:
Command...
Restore state
Trusted state
OK
Cancel
Wait
Exit XFM
Help

```
xfm -client -calleditor -restore
```

Figure 3.1: The Restart Process Box

- The ***Command*** pushbutton pops a File Selector for choosing a different XFace-Maker executable.
- The ***Restore state*** and ***Trusted state*** buttons toggle the -restore and -trusted options (explained below) respectively in the command line.
- ***OK*** launches the specified command line. XFaceMaker will then wait until the new design process connects to it.
- ***Cancel*** will not launch any design process. The user will then be warned that XFaceMaker is running in mono-process mode. Clicking 'No' at this time will go back to the command line dialog box.

- The ***Wait*** button will not launch any design process, but XFaceMaker will still wait for a connection (the XFaceMaker interface is disabled until the connection occurs). This is useful if you want to launch the design process manually from a shell, or by any other external means. In this case, don't forget to set the -client option when launching the design process.

3.1.2 Restoring the Design Process State

The XFaceMaker process keeps a copy of the state of the Design process. The -restore option informs the starting Design process that it must fetch this state from the XFaceMaker process, and modify its own state accordingly. In short, the new Design process will restart where the old one had stopped. The Design process state includes:

- the whole hierarchy of widgets,
- the sets of loaded groups, templates and classes,
- the current editing mode (Try or Build),
- various internal informations: current directory, current file name, current edited object, template or class model objects in edition.

If the Design process was restarted by the user, or if it terminated due to an execution error in Try mode, the Design process state is saved just before the Design process exits.

If the Design process crashed during a widget modification (i.e. when the user clicked ***Rebuild*** or ***Set Values*** in an editing box after modifying a resource or another attribute), then XFaceMaker restores the state of the Design process "just before" the widget was re-built, and things will be as if the user had just made the modifications, but not yet applied them. In this case, the modifications should be corrected and another ***Rebuild*** or ***Set Values*** should be attempted, or the modifications can be discarded by selecting another object (the 'Resource modified/ignore changes?' warning will be issued).

It may happen that the Design process state cannot be saved before it exits, for example if its data is too corrupted. For this reason, XFaceMaker
keeps a second security state, called the 'trusted state' which, by default, is saved every minute, and each time the user switches to Try mode. (You can change the default value using the ***Auto-Save State*** item in the **Options** menu). This state can be restored by using the -restore and -trusted simultaneously. So, the typical scenario is to try first -restore alone (the default), and if this fails try -restore -trusted.

In addition to having the Design process state kept by the XFaceMaker process, it is possible to save the state to disk. This can be useful, for example, to protect against a power failure without saving a .fm file. The '***Save State***' command in the 'File' menu saves the current Design process state into a set of files starting with .xfm_ in the current working directory of XFaceMaker. Once saved, this state can be restored by passing the -restore (and perhaps -trusted) option(s) to the XFaceMaker process. The state will first be loaded from disk files to the XFaceMaker process, and then restored into the initial Design process.

3.1.3 Launching a New Design Process

The user can control the Design process using the ***Restart Design Process*** command in the 'File' menu. This command terminates the Design process (with a confirmation if the current interface was modified). The Design Process Restart dialog box appears. See the previous sections on how to choose the Design process and its options.

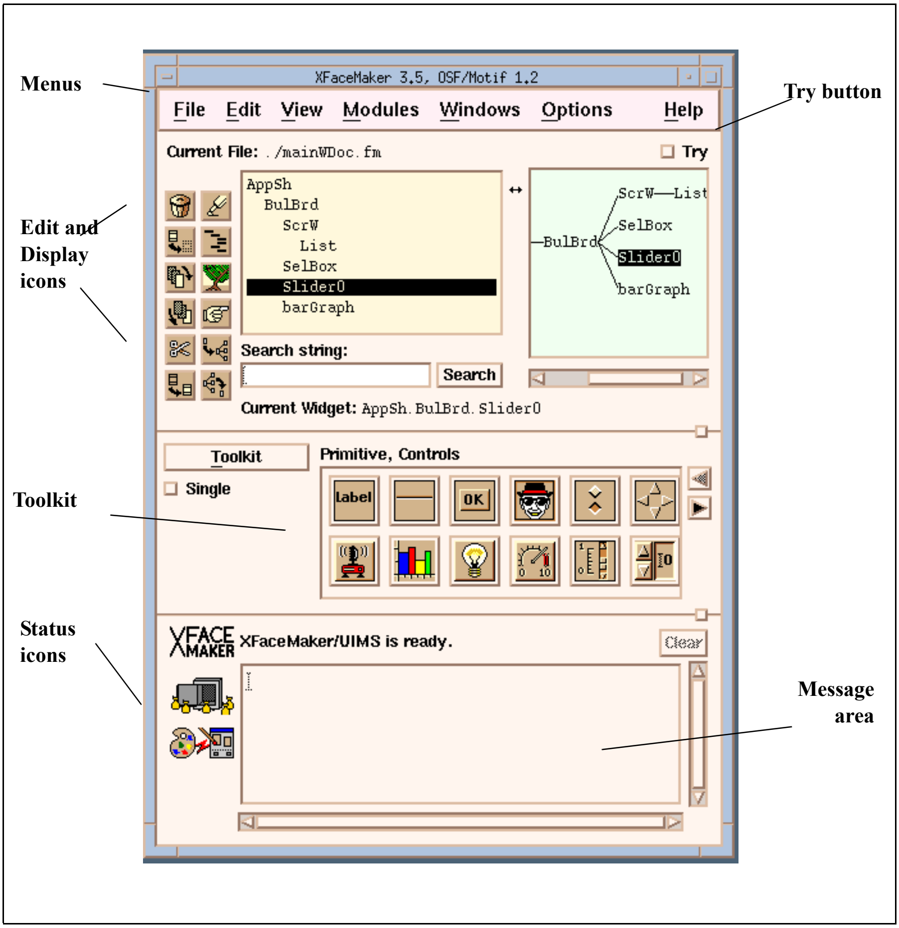
The ***Restart Design Process*** command is used to start a different Design process, and gives access to the 'Dynamic Extension' functionality in XFaceMaker. For example, suppose you want to define a new C function and add it to XFaceMaker so that it can be called in a callback. As in previous versions, you have to write your function, compile it, and link it with a main program containing calls to FmInitialize, FmAttachFunction and FmCallEditor.

In fact, XFaceMaker will do everything – except writing the function – automatically, as explained in the "Application" chapter of the *XFaceMaker User's Guide*. The difference is that you don't have to exit your current XFaceMaker and start the new one from the beginning. Instead, you just have to call the ***Restart Design Process*** command and specify the new program as the Design process. Your new function is now available.

When XFaceMaker is running in mono-process mode (without any Design process connected), the ***Restart Design Process*** switches back to dual-process mode by starting a new Design process.

CHAPTER 4: The Command Window

This chapter describes the XFaceMaker Command window: the Menus, Icons, Try button, Toolkit, and Message area.



Current File: ./mainWDoc.fm

```
AppSh
  BulBrd
    ScrW
      List
    SelBox
    Slider0
    barGraph
```

Search string:

Current Widget: AppSh.BulBrd.Slider0

XFaceMaker/UIMS is ready.

Figure 4.1: The Command Window

4.1 The XFaceMaker Menus

The Command window contains seven pulldown menus: File, Edit, View, Modules, Windows, Options, and Help. Each menu contains a number of commands that you can issue by selecting the appropriate menu item. If a menu item is followed by an ellipsis or an arrow, clicking on the item will open a dialog box or a sub-menu with further choices.

4.1.1 The File Menu

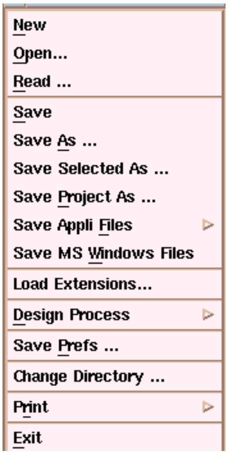


Figure 4.2: The File Menu

The File Menu contains the following items:

- ***New*** Clears XFaceMaker by removing all user-defined widgets from the interface. If you have not saved your work since the last modification, an alert box will warn you. This command also clears any global functions (FACE or application) memorized by XFaceMaker in the course of the session.
- ***Open...***Pops up the File Selector to specify which interface or project file to load. The design process (current interface) is cleared before the new interface is loaded. If you have not saved your work since the last modification, an alert box will warn you. This command does not clear global functions that have been memorized; this allows different interface files to share the same global functions without having to redeclare them.
- ***Read ...***Pops up the File Selector box to specify which interface file to concate-

nate to the current interface. XFaceMaker first checks that the specified file is not already loaded. If so, an error message is issued.

- ***Save*** Saves the current interface under the current file name; that is, the name specified during the last ***Save*** or ***Save As***. The default name is unnamed.fm. That is the name that will be used to save your interface if you have not already saved it with the ***Save As*** command.
- ***Save As ...*** Pops up the File Selector and saves the current interface under the name specified there. The file you name becomes the current file. If a file with the same name already exists in the directory, an alert box will ask your permission to overwrite it.
- ***Save Selected As ...*** Pops up the File Selector and saves the currently selected widget hierarchy into the file you specify. Subsequent ***Save*** or ***Save As*** commands will save the hierarchy to this file.
- ***Save Project As ...***Pops up the File Selector and saves the main and secondary files of an interface as a project (.fmp) file. A project file is a FACE file which contains commands to load the main and secondary files of the interface.
- ***Save Appli Files*** Opens a cascade menu used to select application files that XFaceMaker will generate.

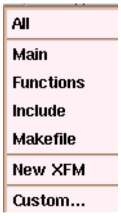


Figure 4.3: The Save Appli Files Menu

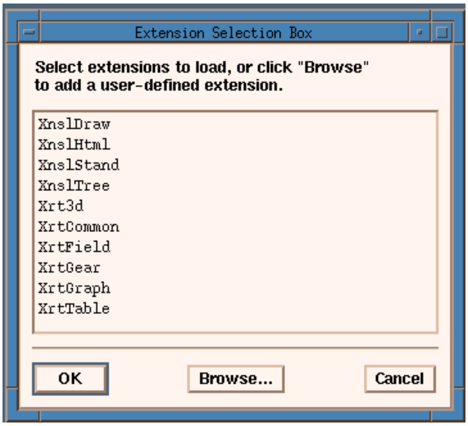
The available options are:

m ***All*** Generates the four next files listed below--Main, Functions, Include, and Makefile-- needed to build an application. If you have made a project file for your interface, you should generate the application files for the project (first load the .fmp file), not just for an individual .fm file of your project. Before generating the application files, make sure that you have set the ***Preserve User Code*** flag as you want it, and the proper C code generating options, as well as the required UIL, RDB, Message Catalog flags. Please refer to Section 4.1.6, The Options Menu.

- m ***Main*** Generates a C file called <*Name*>Main.c, containing the main function of the application. <*Name*> is the name of the .fmp or .fm file that is loaded.
- m ***Functions*** Generates a C file called <*Name*>App.c, containing the skeletons of the application functions as well as the *global* active values. <*Name*> is the name of the .fmp or .fm file that is loaded.
- m ***Include*** Generates aC include file called <*Name*>App.h, containing the declarations of the application functions and global active value variables. <*Name*> is the name of the .fmp or .fm file that is loaded.
- m ***Makefile*** Generates a Makefile called <*Name*>Make, containing the rules to build the application in interpreted, compiled and editor mode. <*Name*> is the name of the .fmp or .fm file that is loaded.
- m ***New XFM*** Generates a C file called <*Name*>XfmMain.c containing the main function of a new instance of XFM, and also a Makefile called <*Name*>Xfm-Make. You can complete the <*Name*>XfmMain.c file as required for your particular needs, or use the ***Load Extensions*** menu item instead, to build a new XFaceMaker executable that includes the functions and widgets that you want to make known to XFaceMaker. See the *User Guide* for further detail.
- m ***Custom*** Pops up a File Selector for choosing a pattern file from which to generate an application file.

- ***Save MS Windows Files*** Generates the Windows version of the interface. Conditioned by the XFaceMaker/Win option in the license file.This command pops a dialog box that gives information on the conversion, in particular on the extent to which the conversion could be achieved.
- ***Load Extensions...*** Pops up a dialog box with a list of extensions available. You can choose the ones you want to include in the new XFaceMaker you will build. To load your own extensions, click on the ***Browse...*** button. This opens a file selector which lets you specify the extension file you want to load. See the *User*

Guide for further detail.



Select extensions to load, or click "Browse"
to add a user-defined extension.

XnslDraw
XnslHtml
XnslStand
XnslTree
Xrt3d
XrtCommon
XrtField
XrtGear
XrtGraph
XrtTable

Figure 4.4: The Extensions Selection Box

- ***Design Process*** The Design Process is the process in which you build your interface interactively, under the control of the XFaceMaker Process - see Section 3.1.1, Restarting the Design Process. This cascade menu provides the following options for saving or re-launching the design process:

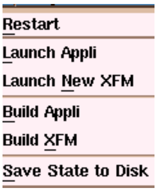
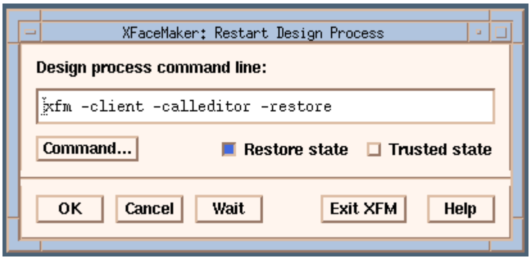


Figure 4.5: The Design Process Menu

m ***Restart*** Terminates the design process. An alert box first asks for confirmation if the current interface has been modified. The Restart Design Process box then appears. You can set the options and restart the design process, cancel, or exit XFaceMaker. If you cancel, an alert box will ask if you want to run in mono-process mode. When XFaceMaker is running in mono-process mode, (without any Design process connected), selecting ***Restart*** switches back to

Dual Process mode.


XFaceMaker: Restart Design Process
Design process command line:
Command...
OK
Cancel
Wait
Exit XFM
Help

```
xfm -client -calleditor -restore
```

☑ Restore state ☐ Trusted state

Figure 4.6: The Restart Design Process Box

- m ***Launch Appli*** Launches the editable version of the application built using the ***Build Appli*** command. The application is launched as a new Design process.
- m ***Launch New XFM*** Like ***Launch Appli***, but launches a new XFM built using the command ***New XFM*** described above.
- m ***Build Appli*** Builds the editable version of your application directly from XFaceMaker. The output is displayed in a pop-up window. You may have to modify the generated Makefile to compile and/or link your application if your system requires special compilation flags or libraries. You can also modify the application file patterns if you want to change permanently the way XFaceMaker generates the application files on your system.
- m ***Build XFM*** Like the ***Build Appli*** command, but used to compile a new XFM after using the command ***New XFM*** described above.
- m ***Save State To Disk*** Saves the current state of the Design process in the current directory in a set of files prefixed by .xfm_. Used to save the process to disk. Use with the -restore option.

- ***Save Prefs ...*** Saves your current screen configuration, including the position of the widgetstores, editing boxes, etc., as well as the currently loaded groups, templates, and classes and the options set in the ***Options*** menu. This information is stored in a preference file called .xfm.startup in your home directory. XFaceMaker loads this file automatically on startup. Note that none of the user interface windows are memorized by this command.
- ***Change Directory...*** Pops up the File Selector for specifying a different working directory for interface and application files.
- ***Messages*** Only on internationalized version of XFaceMaker, i.e.on the machines that have GLS (Global Language Support). Opens a cascade menu containing the

following items:

m ***Open Catalogs...*** Pops up the File Selector to specify the name of the message catalog from which internationalized messages should be fetched. Several message catalogs can be opened successively. XFaceMaker then looks for a message in all the open catalogs, starting with the last one opened.

m ***Close Catalogs*** Closes all the open message catalogs.

- ***Print...*** Opens a cascade menu with the following items:

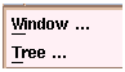


Figure 4.7: The Print Menu

m ***Window...*** Used to create a postscript file of an interface window, using the xwd command. When you choose ***Print,*** a cross hair cursor appears. Click on an interface window to select it. The File Selector box pops up in which to name the postscript file. By default, the .ps file name is that of the window's shell widget. Print options are defined in the Print Options box.

m ***Tree...*** Pops up the File Selector for specifying the postscript file in which to print the widget tree of the interface currently loaded. Print options defined in the Print Options box have no effect on this command.

- ***Exit*** Exits from XFaceMaker. If you have not saved your work since the last time you made a change, an alert box will warn you.

4.1.2 The Edit Menu

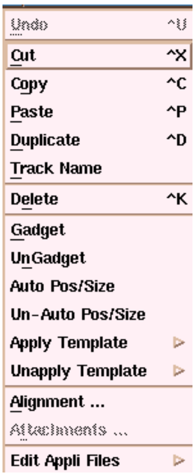


Figure 4.8: The Edit Menu

The Edit Menu contains the following items:

- ***Undo*** Undoes the effect of the immediately preceding edit command. When there is no command to *undo*, this line is dimmed to provide the visual cue that clicking in it will have no effect.
- ***Cut*** Cuts the current widget selection from the interface and places it in the widget cut buffer. These widgets can be retrieved from the widget cut buffer with the ***Paste*** command.
- ***Copy*** Copies the current widget selection and places it in the widget cut buffer. These widgets can be retrieved from the widget cut buffer by means of the ***Paste*** command.
- ***Paste*** Pastes the contents of the widget cut buffer at the designated point in the interface. Or, using the Widget Tree or the Widget List, at the specified place in the widget hierarchy.
- ***Duplicate*** Duplicates the current selection, and places an exact copy slightly to the right and below the original. The selection is not placed in the widget cut buffer and therefore cannot be retrieved with the ***Paste*** command.
- ***Track Name*** Places in the X Buffer (X selection that is pasted with Mouse button

2) the widget path of a widget relative to the current widget. The cursor changes to a question mark. Drop it on the widget whose path you want to get, then paste with mouse button2 in a text input window. You can obtain the absolute path of a widget by clearing the widget selection.

- ***Delete*** Deletes the current widget selection. The selection is not placed in the widget cut buffer and therefore cannot be retrieved with the ***Paste*** command.
- ***Gadget*** Changes into gadgets all the widgets that are in the sub-tree of the selected widget. Only the widgets that have a gadget equivalent and in which no widget-specific resources are used are changed. The number of widgets changed is displayed in the Status field. Note that widgets can also be changed to gadgets individually, by using the *Gadget* toggle button in the Resources box.
- ***UnGadget*** Changes into widgets all the gadgets that are in the sub-tree of the selected widget. The number of widgets changed is displayed in the Status field. Note that gadgets can also be changed to widgets individually, by using the *Gadget* toggle button in the Resources box.
- ***Auto Pos/Size*** Removes unnecessary position and size resources from the widgets that are in the sub-tree of the selected widget. The resource definitions are removed from the .fm file where other resources or parent widget resources override the values, making them redundant. For example, an XmList widget's x and y resources are redundant if its parent is an XmScrolledWindow, because the parent will define its position. The number of widgets changed is displayed in the Status field. Widgets can also be changed individually, by using the *Auto Pos* and *Auto Size* toggle buttons in the Resources box.
- ***Un-Auto Pos/Size*** Inserts all the currently undefined position and size resources for widgets that are in the sub-tree of the selected widget in the interface description. All x, y, height and width resources are defined, even when they are not required because another resource, or another widget's resources, define the actual values used.
- ***Apply Template*** Applies a specified template to the currently selected object. Opens a cascade menu for selecting the widget hierarchy to which the template will apply. You can apply a template to:

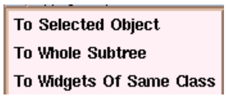
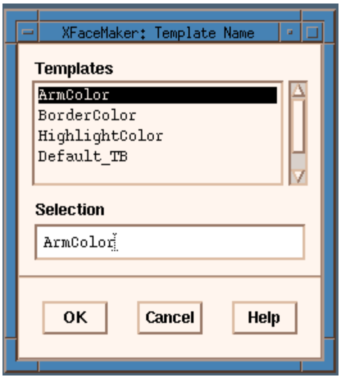


Figure 4.9: The Template Menu

- m ***To Selected Object*** Applies the specified template to the selected object.
- m ***To Whole Subtree*** Applies the specified template to the selected object and its entire subtree.
- m ***To Widgets of Same Class.*** Applies the specified template to the selected object and all widgets of its class in the subtree.

The Template Name box is popped up to specify which template should be applied.



```
ArmColor
BorderColor
HighlightColor
Default_TB
```

Selection: ArmColor

Figure 4.10: The Template Name Box

- ***Unapply Template*** Opens the template menu to specify the widget hierarchy from which the template will be removed. You can unapply a template:
  - m ***To Selected Object*** Removes a specific template from the currently selected object.
  - m ***To Whole Subtree*** Removes a specific template from the currently selected object and its entire subtree.
  - m ***To Widgets of Same Class.*** Removes a specific template from the currently selected object and all widgets of its class in the subtree.

  The Template Name box pops up to specify which template should be removed.
- ***Alignment...*** Pops up the Alignment box, to align widgets relative to each other or to their parent widget. Widgets can also be snapped to the nearest point on a predefined grid. Grid spacing can be adjusted using the slider. The grid can be activated only if the current widget is a Composite. Refer to Section 5.2, The

Alignment Box.

- ***Attachments...*** Pops up the attachment box in the resource editing window. This item is sensitive only if the current widget is a child of XmForm. The attachment box is popped up in the same way when you click in the ellipsis of an attachment resource in the Resource editing window.Refer to Section 5.3, The Attachments Box.
- ***Edit Appli Files*** Opens a cascade menu with a list of application files that can be edited. Selecting one pops up an editing window in which you can modify the specified file. The type of editing window that is opened (vi, emacs, Wx, etc.) is controlled by the **FMEDITOR** environment variable.

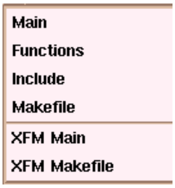


Figure 4.11: The Edit Appli Files Menu

4.1.3 The View Menu

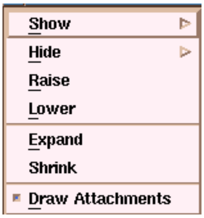


Figure 4.12: The View Menu

The View Menu contains the following items:

- ***Show*** A cascade menu provides the following choices:

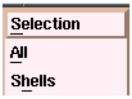


Figure 4.13: The View Cascade Menu

  m ***Selection*** Makes the current selection visible.

  m ***All*** Makes all current widgets visible, selected or not.

  m ***Shells*** Makes all shell widgets visible, selected or not.

- ***Hide*** When the interface is saved, only those widgets that are not *hidden* will be marked as mapped in the file; they will be visible when the interface is instanciated by the application. Before saving your interface, you should hide any widgets you do not want visible when the application starts up. A cascade menu provides the following choices:

  m ***Selection*** Hides the current selection.

  m ***All*** Hides all widgets on the interface, whether selected or not.

  m ***Shells*** Hides all shell widgets.

- ***Raise*** Moves the current widget selection to the foreground, where it no longer is occluded by other widgets. This does not modify the interface file, just the current display.
- ***Lower*** Moves the current widget selection to the background where it will no longer occlude other widgets. This does not modify the interface file, just the current display.
- ***Expand*** In the Widget Tree or Widget List, displays the widgets that belong to the subtree of the selected widget, i.e. expands a shrunk node.
- ***Shrink*** In the Tree or List, hides any widgets that are descendants of the selected widget, i.e. reduces the subtree to a node: the selected widget.
- ***Draw Attachments*** Specifies that widget attachments be visible. This does not modify the interface file, just the current display.

4.1.4 The Modules Menu

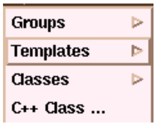


Figure 4.14: The Modules Menu

The Modules Menu contains the following items:

- ***Groups*** Opens a cascading menu containing:
  - m ***Load...*** Pops up the File Selector box to specify which group to load in the Groups widgetstore.
  - m ***Save As..***. Saves the selected widget and its subtree as a group, under the name specified in the file selector box, in a .fm description file for later reloading, and adds the group to the Groups widgetstore.
  - m ***Delete...*** Deletes the specified group from the Groups widgetstore. This does not destroy the .fm group description file saved with ***Save As...*** .
  - m ***Choose Icon...*** Pops up the Images box to select or create an image for the icon that will represent the specified group in the widgetstore.
- ***Templates*** Opens a cascading menu containing:

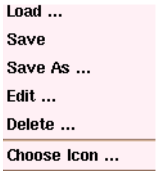


Figure 4.15: The Templates and Classes Cascade Menu

  - m ***Load...*** Pops up the File Selector box to specify which template to load in the Templates widgetstore.
  - m ***Save...*** Saves the selected object and its subtree as a template in a .fm description file, and creates a default icon for the object in the Templates widgetstore. If the selected object is not the same as the object selected during the last ***Save As*** or ***Edit***, XFaceMaker pops a confirmation dialog.
  - m ***Save As...*** Saves the selected object and its subtree as a template under the

specified template name and .fm file name. A default icon is added to the Templates widgetstore to represent the newly saved template.

- m ***Edit...*** Pops up the Template Name box in which to specify a template name. The template specified is then mappped and selected for editing.
- m ***Delete...*** Deletes the template specified in the Template Name box. This does not delete the .fm file saved with ***Save*** or ***Save As...***
- m ***Choose Icon...*** Pops up the Icon Selection box to select or create an image for the specified template's widgetstore icon. This box works like the box described in section 5.12, The Menu Editor.

• ***Classes*** Opens a cascading menu containing:

- m ***Load...*** Loads the .fm widget class definition file specified in the file selector box, and adds a default icon to the widgetstore.
- m ***Save...*** Saves the selected object and its subtree as a class in the last widget class definition file specified with ***Save As, Load,*** or ***Edit***, and with the same widget class name. If the selected object is not the same as the object selected during the last ***Save As*** or ***Edit***, XFaceMaker pops a confirmation dialog.
- m ***Save As..***. Used to save the selected object and its subtree in a widget class definition file. Pops the File Selector box to specify the file in which the selected object is to be saved. Next, pops up the Class Name box, to specify the name of the new class. Class names *and* class file names must start with an uppercase character.
- m ***Edit...*** Reads a widget class definition file as a regular .fm file, maps it in a shell and selects its top-level object.
- m ***Delete...*** Deletes the user-defined class specified in the Class Name box, and removes its icon from the widgetstore. Note that the corresponding widget class is not actually deleted, because the XToolkit provides no support for this action. The instances of the class that you have created are not destroyed. This does not delete any previously saved .fm file.
- m ***Choose Icon..***. Pops up the Icon Selection box to select or create an image for the widgetstore icon of the class. This box works like the box described in section 5.12, The Menu Editor.

• ***C++ Class*** Pops up a special File Selector for saving the selected object and its subtree as a C++ class. This is described in section 5.6, C++ Class Name Box.

4.1.5 The Windows Menu

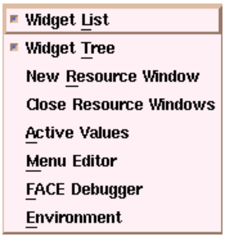


Figure 4.16: The Windows Menu

The Windows Menu contains the following items:

- ***Widget List*** Opens/closes the Widget List box, displaying the hierarchical list of widgets of the current interface in the XFaceMaker main window. See section 4.5, The Widget Tree and Widget List. The widget list box status can be toggled also with the List and Tree icons.
- ***Widget Tree*** Opens/closes the Widget Tree box, which displays the current interface hierarchy as a tree structure. See section 4.5, The Widget Tree and Widget List. The widget tree box status can be toggled also with the List and Tree icons.

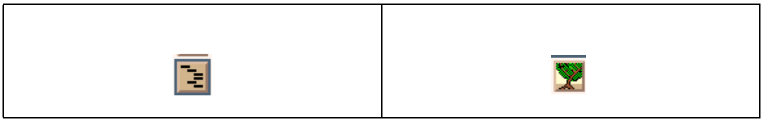


Figure 4.17: The List and Tree Icons

- ***New Resource Window*** Opens a Widget Resources editing window. By default, the resources of the currently selected widget are displayed in the window. If no widget is selected, the fields will be empty. The popup menus on widgets in the Design Process also let you open a resource window and/or change the current widget in the resource window.
- ***Close Resource Windows*** Closes all open Widget Resource windows.
- ***Active Values*** Opens/closes the Active Values box, which allows you to define active value names and scripts. Active values are usefull
  - to share data between the interface and the application
  - to add pseudo resources to a widget
  - to define a drag source

    - to define a drop site
    - to define the resources of a new widget class
    - to define the data members of a new C++ class

  See section 5.1, The Active Values Box and the *XFaceMaker User Guide* for further detail.

- ***Menu Editor*** Opens/closes the Menu Editor box, for building menus interactively. See section 5.12, The Menu Editor.
- ***FACE Debugger*** Opens the FACE Debugger which you can use to debug the FACE scripts that you write. See section 5.8, The FACE Debugger
- ***Environment*** Opens the Environment window for setting environment variables. See section 5.7, The Environment Box.

4.1.6 The Options Menu

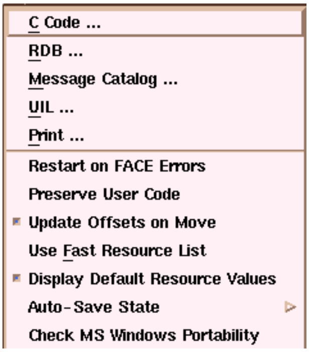


Figure 4.18: The Options Menu

The Options Menu contains the following items:

- ***C Code...***Pops up the C Code Options window. When the ***Save C Code*** checkbutton is set and applied in this window, XFaceMaker generates a C language version of the interface each time the interface is saved, according to the options set in the window. C-code generated by XFaceMaker varies according to the options that have been set in the C Code Options window, and also according to the settings of the other options such as UIL, RDB, etc... See section 5.4, C Code Options and the *XFaceMaker User's Guide* for further detail.
- ***RDB...*** Pops up the RDB Options window. When the ***Save RDB file*** checkbutton is set and applied, some or all of the resources of the interface will be saved into an RDB (Resource Data Base) file when the interface is saved, as well as a version of the .fm file without the RDB resources, whose extension is fm_rdb. Use the Save RDB Options window to specify RDB options, i.e. the types or names of resources that are to be copied to the RDB file and stripped from the .fm_rdb file. See section 5.15, RDB Options for further detail.
- ***Message Catalog*** Pops up the Message Catalog Options window. When the ***Save message catalog file*** checkbutton is set and applied, a message catalog source file will be saved when the interface is saved. Un-numbered messages are numbered automatically. Use this window to specify the default set number used for internationalized strings. See the *XFaceMaker User's Guide*, Internationalization chapter and section 5.13, Message Catalog Options Box for further detail.
- ***UIL...*** Pops up the UIL Options window. When the ***Save UIL file*** checkbutton is

set and applied, a UIL language version of the interface will be generated when the interface is saved. See section 5.20, UIL Options and the *XFaceMaker User's Guide* for further detail.

- ***Print...*** Pops up the Print Options box for specifying how PostScript files will be generated for printing an interface window. See section 5.14, The Print Options Box.
- ***Restart on FACE Errors*** When set, (indicated by a checkbutton), the Design process will be restarted when an error is encountered in a FACE script. In that case, the Command Prompt window is popped up, allowing you to re-launch the Design process. See section 3.1, Error Protection and the XFaceMake User's Guide.
- ***Preserve User Code*** When set, indicated by a checkbutton, the pattern information is kept (as comments) in the application files generated by XFaceMaker. This makes it possible to modify and later re-generate the application files. See the section on Modifying Application Files in the *XFaceMaker User's Guide*.
- ***Update Offsets on Move*** When set, the offsets of a widget that is attached (child of an XmForm) will be updated when the widget is moved. If this checkbutton is not set, XFaceMaker will not let you move the widget interactively.
- ***Use Fast Resource List*** When set, any new Resource windows you open will be displayed with the "Fast" list of resources; that is, without icons, without the popup men and without the resource drag and drop capability. Use this option if your machine is slow in refreshing the resource list when you change the current widget.
- ***Display Default Resource Values*** When set (the default), XFaceMaker tries to display the default values of the unset resources in the Resource window's resource list. This means that an XtGetValues call is performed for every resource, and the value is then converted to a string if possible. If this option is not set, default values are left blank for unset resources, and are displayed only when the user clicks on a particular resource line and/or pops the specialized resource editor. Refreshing the resouce window is faster if this checkbutton is not set. Unset this toggle if you don't need to know the value of unset resources and/or if your machine is slow in refreshing the resource list.
- ***Auto-Save State*** This sub-menu controls how often the *trusted state* of the Design Process is saved automatically by XFaceMaker. The default is ***every minute***. Selecting ***never*** diables the auto-save mechanism. In that case, the Design Pro-

cess state is only saved when the user switches to Try mode.

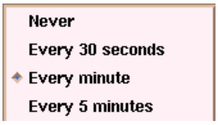


Figure 4.19: The Auto Save State Cascade Menu

- ***Check MS Windows Portability*** When this checkbutton is set, XFaceMaker will verify each resource setting within the interface being loaded or edited, for Windows compatibility. When a resource that XFaceMaker finds a resource setting that it will not be able to translate to Windows code, an error message is output in the main window message area. Moreover, XFaceMaker places any resources that are not compatible with MS Windows in the *Restricted Resources* of the Resource box so as to prevent you from setting such resources when developing an interface that you will want to convert to MS Windows.

4.1.7 The Help Menu

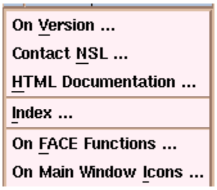


Figure 4.20: The Help Menu

The Help Menu contains the following items:

- ***On Version...*** Pops up a box with information about the version of XFaceMaker currently running.
- ***Contact NSL ...*** when clicked, connects your Netscape browser to NSL's www site.
- ***XFaceMaker Tutorial ...*** launches Adobe Acrobat Reader to view the tutorial.
- ***XFaceMaker Reference Manual ...*** Launches Adobe Acrobat Reader to view this Reference Manual.
- ***XFaceMaker User Manual ...***. Launches Adobe Acrobat Reader to view the User Manual.
- ***Motif 2.1 User Manual ...*** Launches Adobe Acrobat Reader to view the O'Reilly User Manual for Motif 2.1.
- ***Motif 2.1 Reference Manual ...*** Launches Adobe Acrobat Reader to view the O'Reilly Reference Manual for Motif 2.1.

You may find further Help menu items added by your System Administrator giving additional *local* information on using XFaceMaker. See Chapter 16 on how to add Help items.

4.2 The Edit/Display Icons

There are three sets of icons in the Command window to help you visualize and edit your interface. The Edit/Display icons are described below. The Status icons are described in Section 4.7. The third set of icons represent the widgets in the Toolkit.The Toolkit is described in Section 4.6.

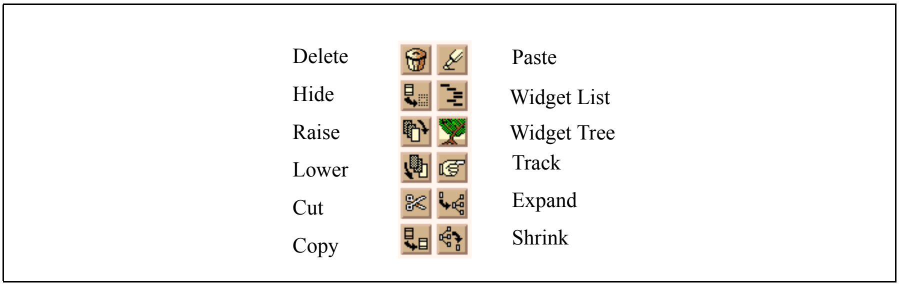


Figure 4.21: The Edit/Display Icons

The edit icons are briefly described below. For more details on their use, please consult the *XFaceMaker User's Guide*.

- ***Delete*** Used to delete widgets from the interface.
- ***Hide*** Used to hide selected widgets on an interface.
- ***Lower*** Used to move selected widgets to the background so that they no longer occlude other widgets.
- ***Raise*** Used to move selected widgets to the foreground so that they are no longer occluded by other widgets.
- ***Cut*** Used to remove the selected widgets from the interface and put them into the widget cut buffer.
- ***Copy*** Used to place a copy of the selected widgets into the widget cut buffer. The widgets are not removed from the interface.
- ***Paste*** Used to place the current contents of the widget cut buffer in a window.
- ***List*** Used to display or close the Widget List.
- ***Tree*** Used to display or close the Widget Tree.
- ***Track*** Changes the mark cursor to a question mark, waits for the use to click on a widget or widget name, gets the name of that widget in the interface hierarchy, stores it in the X buffer for pasting, and displays it in the status indicator field of the main command window. If no widget is currently selected, the tracked widget's absolute name is displayed. If there is a currently selected widget, the

tracked widget's name *relative* to the *selected* widget is displayed. The selected widget name can then be pasted in an XFaceMaker text field.

- ***Expand*** Used to display all sub-trees of the selected widget in the Widget Tree or List.
- ***Shrink*** Used to hide all sub-widgets of the selected widget in the Widget Tree or List.

4.3 The Edit/Select Popup Menus

Pressing Mouse Button 3 on a widget in the Design Process, or on its name in the Widget List or in the Widget Tree opens a Popup Menu whose contents changes with the context: the menu can have up to three main categories as follows.

- ***Selection***

This set of items is present only if the widget selection is not empty, i.e. at least one widget in the interface is selected. It provides a shortcut to issue commands that are also available through the ***Edit*** pulldown menu and that apply to the current selection. When the menu is popped, commands that cannot be issued are shown insensitive, e.g., ***Paste*** when the widget cut buffer is empty.

m ***Cut*** Remove the selected widgets from the interface and place them into the widget cut buffer.

m ***Copy*** Place a copy of the selected widgets into the widget cut buffer. The widgets are not removed from the interface.

m ***Paste*** Place the current contents of the widget cut buffer in a window.

m ***Duplicate*** Duplicate the current selection and place an exact copy slightly to the right and below the original. The widget cut buffer's content is not modified.

m ***Delete*** Delete the current selection from the interface. The widget cut buffer's content is not modified.

- ***Current Object***

This set of items is present only if there is a current object, i.e. if the widget selection is not empty. It has items that let you interact with the selected widget, or select its parent.

m ***Rebuild*** Destroy the widget and build a new one with the resource settings currently defined for the widget.

- m ***Select <name>'s parent*** where <*name*> is the name of the current widget. Select the widget's parent, i.e. move the selection one step up in the widget tree.
- m ***Resources for <name>*** where <*name*> is the name of the current widget. If a resource window is open and its Current selection toggle is set, display the resources for widget *name* in the window. Otherwise, open a new resource window, with widget *name*'s resources.
- m ***Auto Size*** Let the current widget take its default size, i.e. remove any width/height resource settings from the interface description.
- m ***Auto Position*** Let the current widget take its default position, i.e. remove any x/y resource settings from the interface description.

• ***Cursor Object***

This set of items is present only if the widget under the cursor is not the current object. It has items that change the current selection and/or open a Resources Box.

- m ***Select <name>*** where <*name*> is the name of the widget under the cursor. Clear the current selection and make widget *name* the current widget.
- m ***Resources for <name>*** where <*name*> is the name of the widget under the cursor. Add widget *name* to the selection, make it the current widget. If a resource window is open and its Current selection toggle is set, display the resources for widget *name* in the window. Otherwise, open a new resource window, with widget *name*'s resources.

4.4 Try Mode and Build Mode

XFaceMaker has two modes of operation, *Try* and *Build*. Use:

- *Build mode* when you are building or editing the interface. Events you generate in the Design Process are received by the XFaceMaker process. You can select widgets by clicking on them in the design process windows.
- *Try mode* when you want to test some aspect of the interface to see how it will work in the final application. Events you generate in the Design Process are received by the widgets in your Design Process. You can activate your callback scripts.

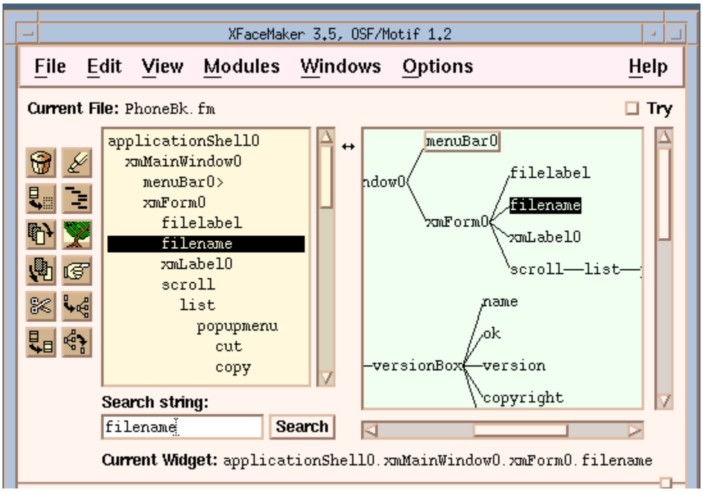
There are a number of editing actions that cannot be performed in Try mode. Does the cursor become a caution pointer? Whenever you have trouble carrying out some editing action in XFaceMaker, check first to see if you have forgotten to switch back to Build mode.

To put the editor into Try mode, click on the Try button in the Main window---the button is checked when selected.

To return to Build mode, de-select the Try button.

4.5 The Widget Tree and Widget List

The Widget Tree and Widget List allow you to visualize the hierarchical structure of your interface and perform some editing actions. You can display one or both at a time. When both are displayed, an arrow between them allows you to resize the display laterally.



Current File: PhoneBk.fm

Search string: filename

Current Widget: applicationShell0.xmMainWindow0.xmForm0.filename

Figure 4.22: The Widget List and Widget Tree

The Widget List and Widget Tree have the following features:

Current File Displays the name of the current .fm file.

Current Widget Displays the pathname of the currently selected widget, or the last widget selected if more than one widget is currently selected, or nothing if no widget is selected.

Search string A text input field used to specify the first few characters of a widget name that is searched for when the ***Search*** pushbutton is clicked or upon typing *Return*.

Search When clicked, searches for the specified string in the widget names and selects the widget where it first occurs in the Widget List, Widget Tree, and the interface. Click on Search again to find other occurences of the string specified. This is useful to locate specific widgets in large interfaces.

4.5.1 The Widget List

The Widget List displays the widgets in the interface as well as their structural relationships. Widget names are indented from the left to reflect their position in the widget hierarchy. Thus, a child widget is listed below its parent and one character to the right. Currently selected widgets are highlighted.

A vertical scroll bar lets you access all the objects in an interface. Use the ***Shrink*** and ***Expand*** icons to control which subtrees of the Widget List you see.

You can use the Widget List to select widgets or gadgets, to reorder sibling widgets, and to specify where, in the hierarchy, you wish to add a widget. Open/close the box by clicking on the ***Widget List*** item in the Windows menu, or by toggling the Widget List display icon.

4.5.2 The Widget Tree

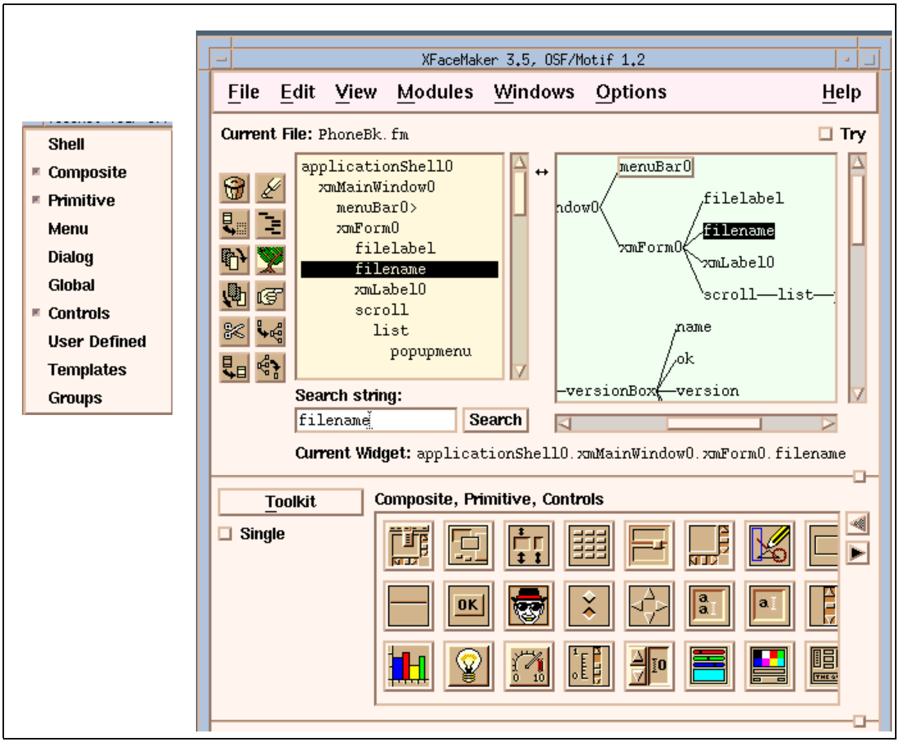
The Widget Tree displays your current interface as a tree structure. Scroll bars allow you to see all the objects in the interface. You can also enlarge the box to get a larger view of the interface tree structure. Use the ***Shrink*** and ***Expand*** icons to control which subtrees of the Widget Tree you see.

You can use the Widget Tree to select widgets or gadgets, to change a widget's name, to reorder siblings (change their creation order) , and to specify where, in the hierarchy, you wish to add a widget. Open/close the box by clicking on the ***Widget Tree*** item in the Windows menu or by toggling the Widget tree display icon.

You can print the Widget Tree of your interface with the ***Print*** button in the ***File*** menu.

4.6 The Toolkit

The Toolkit gives you access to the widgets used to build an interface. Widgets are grouped by kind in “widgetstores”. When a widgetstore is open, its class category is displayed and each of its widgets is represented by an icon. When you move the pointer over an icon, the widget’s class name is displayed to the left of the widgetstore. Use the arrow buttons to the right of the Toolkit to view icons that are not immediately visible.



Current File: PhoneBk.fm

```
applicationShell0
  xmMainWindow0
    menuBar0>
    xmForm0
      filelabel
      filename
      xmLabel0
      scroll
        list
          popupmenu
```

Search string: filename

Current Widget: applicationShell0.xmMainWindow0.xmForm0.filename

Composite, Primitive, Controls

Figure 4.23: The Toolkit

When the “single class” toggle button is set, only one widgetstore can be open at a time. To open more than one, de-select the “single class” toggle button, and then select the categories you wish to open from the ***Toolkit*** pulldown menu. Open widgetstores have a checkmark next to their name in this menu. To close a widgetstore, de-select its name.

To create an instance of a widget, click on the widget's icon—the cursor changes to a cross. Bring the cross over the desktop and drag to draw the desired object. Or, for widgets that size automatically, bring the cursor to the desired parent widget in the interface, the Widget Tree, or the Widget List, and click.

You can apply/unapply a template to a selected widget by pressing mouse Button 3 on the template icon in the widgetstore and selecting the command from the cascade button that pops up. You can apply/unapply the template to: the selected object, the whole sub-tree, or all widgets of the same class.

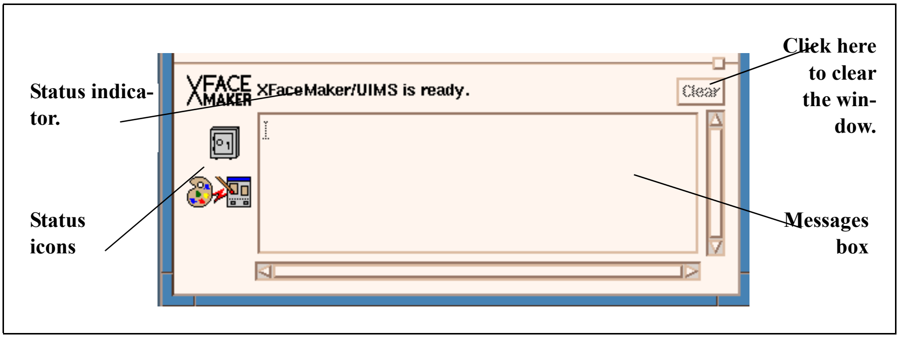
The Toolkit contains the following widgetstores:

- **Primitive** contains the widget types *List, TextField, Label, Drawnbutton, ToggleButton, Text, Separator, PushButton, ScrollBar, ArrowButton*.
- **Composite** contains the widget types *Scale*, *DrawingArea*, *BulletinBoard, RowColumn, MainWindow, Form, Frame, PanedWindow, ScrolledWindow.*
- **Shell** contains the widget types *DialogShell, ToplevelShell, ApplicationShell, OverrideShell, TransientShell.*
- **Menu** contains the widget types *OptionMenu, PulldownMenu, CascadeButton, MenuBar, PopupMenu*.
- **Dialog** contains the widget types *FileSelectionBox, SelectionBox, MessageBox, Command*.
- **Global** contains the Display and Screen global objects defined by the Motif library.
- **Control** contains the NSL Control widget types *XnslJoyStick, XnslBarGraph, XnslIndicator, XnslMeter, XnslSlider, XnslStepper, XnslColorBox, XnslColorSelectionBox, XnslFontSelector.*
- **Group** contains user-created *Group* widgets. This widgetstore is initially empty unless the .xfm.startup file contains commands to load certain groups. Only appears if you have defined a group.
- **User Defined** contains any user defined widgets that have been linked with XFaceMaker, or created with the Widget Class edit box. Only appears if you have defined a widget.

- **Template** contains any user defined templates created with the *Templates* commands in the Modules menu.Only appears if you have defined a template.
- **NSL** contains widgets developed by NSL. Only appears if you have an extended XFaceMaker.
- **NSL DrawGadgets** contains the XDraw gadgets. Only appears if you have an extended XFaceMaker that includes the XDraw library.

4.7 The Message Area

The Message area at the bottom of the Command window displays information on the state of the editor, and messages from XFaceMaker.



XFaceMaker/UIMS is ready.

Figure 4.24: The Message Section

- The Status line above the message box describes the current internal condition of XFaceMaker itself. While you are loading or saving files, this line states which operation is underway, and whether it has succeeded or failed.
- The message box displays various messages from XFaceMaker, including error messages. The message box may be cleared by clicking on the ***Clear*** button on the top right-hand side of the Message area.
- The Message icons are shown below.

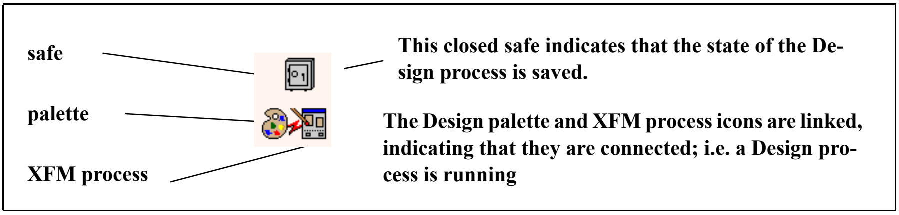

safe
palette
XFM process
This closed safe indicates that the state of the Design process is saved.
The Design palette and XFM process icons are linked, indicating that they are connected; i.e. a Design process is running

Figure 4.25: The Message Icons

CHAPTER 5: The Editing Boxes

This chapter contains brief descriptions of the XFaceMaker dialog boxes and their use. Many of these boxes can be opened by clicking on the appropriate menu item, but some are popped up only when the associated option or resource is selected. The boxes are listed alphabetically, with sub-boxes listed under the main dialog box they are opened from.

Certain boxes are modal; that is, you must either respond to the question posed by the box or close it before you can take any other action in XFaceMaker. If you attempt some other action before responding to a modal box, you will find that you are unable to proceed, and the cursor takes the form of a caution sign to indicate this. To continue, simply enter an appropriate response in the box, or close it.

5.1 The Active Values Box

The Active Values box is used to define and edit the active values of the currently selected widget. Open it by clicking on ***Windows/Active Values***. The options available in this box change according to the usage you select for the active value. There are five choices for ***Usage***:

- ***interface*** The active value is used to share data with the application, or to define a script, a pseudo-resource, or a method.
- ***drag source*** The active value names a target the widget can drag.
- ***drop site*** The active value names a target accepted by the widget as a drop site.
- ***widget class*** The active value is a resource in a widget class described by the widget.
- ***C++ class*** The active value is a member of the C++ class generated in connection with the widget.

There are a number of fields that always appear.

A list of active values Only active values that have the same usage appear in this field, if you are using "new style" active values (version 3.0 and later). However, older .fm files with "old style" active values will display *all* the active values of the widget, regardless of their usage.

Usage Used to specify the usage for the active value.

Custom Editor Opens the editor you specified using the FMEDITOR environment variable.

Clear clears the editing field.

Get/Set Script Used to enter get and set scripts for the active value. By convention, the *get* script should do whatever is necessary to update the active value's data to reflect the interface's current setting. The *set* script should do whatever is necessary to update the interface so that it displays the current value of the data.

Add adds the newly defined active value to the list of active values for the widget.

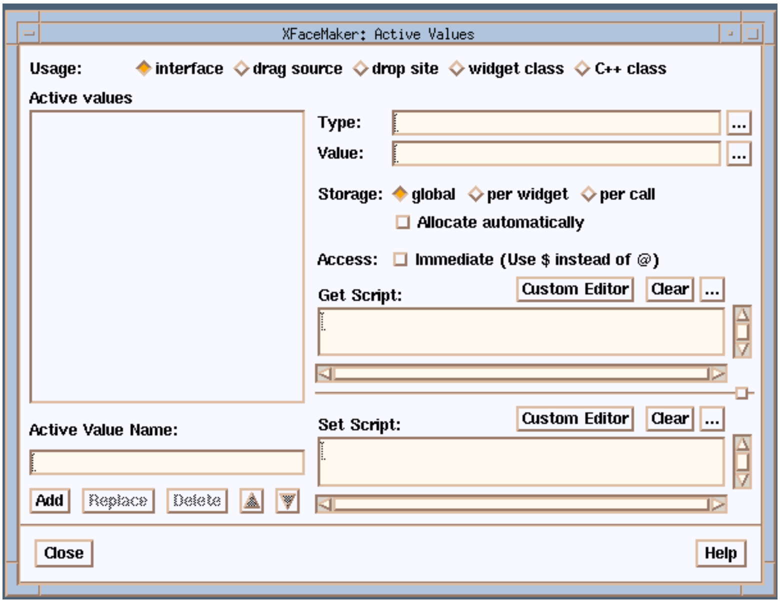
Replace Replaces the active value that is being edited with the modified settings.

Delete Deletes the selected active value and its get and set scripts.

Close Closes the Active Values window. If you have added (by clicking on the ***Add*** button) an active value to the widget you are editing, the active value is registered with the widget. If not, clicking ***Close*** merely closes the window without changing the widget.

5.1.1 Interface

An active value whose usage is set as “interface” is used to share data with the application, or to define a script, a pseudo-resource, or a method.


XFaceMaker: Active Values

Usage: [✓ interface] drag source drop site widget class C++ class

Active values

Type:

Value:

Storage: [✓ global] per widget per call

Allocate automatically

Access: Immediate (Use $ instead of @)

Get Script:

Set Script:

Active Value Name:

Figure 5.1: Interface Active Values

Active Value Name Used to enter a name to identify the active value. The name must begin with an alphabetic character.

Type Used to enter the FACE type of the active value variable. If no type is specified, the type is “Any”. The ellipsis pops up a type selection box. The size of an active value is the size of an XtArgVal.

Value If storage is *per widge*t and allocated automatically by XFaceMaker, you may enter an initial value for the active value here.

Storage Used to specify data storage and how it is allocated.

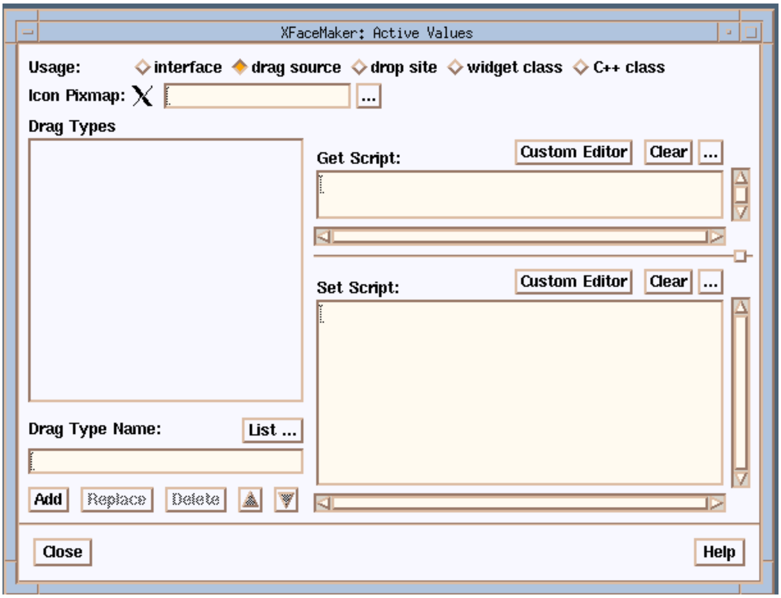
- ***global***: if the entire interface may always use the same data.
- ***per widget***: if you want separate storage locations for each widget in the interface that uses the active value.
- ***per call***: if you want XFacemaker to allocate storage each time it calls the active value’s set script---and then de-allocate it after the script is executed.

Allocate automatically when set, XFaceMaker will allocate storage for the active value unless the application has already allocated storage (i.e., the active value has already been attached to an application variable). With automatic allocation, you can test your active value scripts in ***Try*** mode. If you do not specify automatic allocation, the contents of the active value is not accessible in Try mode. Note that, if you specify a default value for an active value of type String, XFaceMaker will always allocate storage.

Access You must decide whether your FACE script will refer to the contents of the active value via the @ special variable, or by its immediate value using $. If by immediate value, select the checkbutton marked “immediate”.

5.1.2 Drag Source

An active value whose usage is set to "drag source" defines a target the widget can export in a Motif Drag & Drop operation.



Usage: drag source

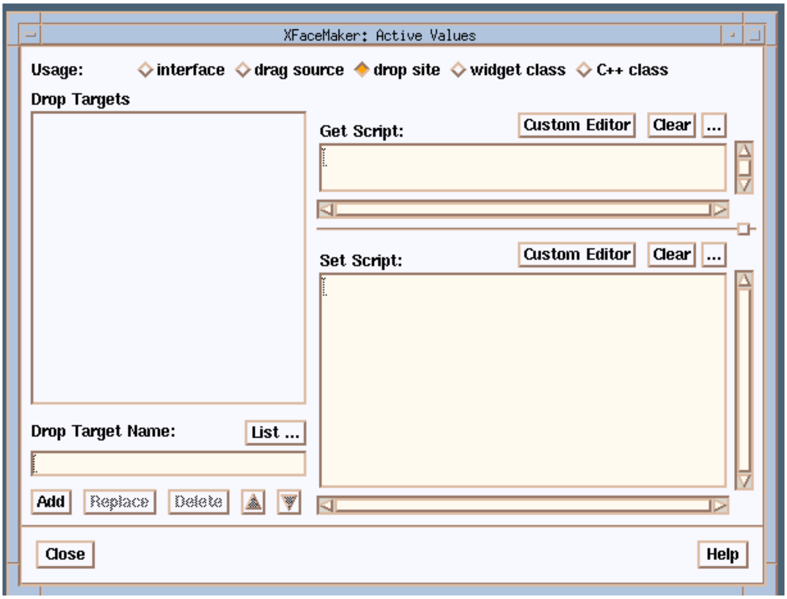
Figure 5.2: Drag Source

Icon Pixmap Used to enter the name of a pixmap that the pointer changes to when you start a drag operation. The ellipsis pops up the Pixmap editor for defining a pixmap.

Drag Type Name Used to enter the type of target that the selected widget can export. The ***List...*** button pops up the Drag Target Selector which contains alist of commonly accepted selection targets defined in the ICCCM. Select a drag target from the list, then click on the ***OK*** button. A widget can have more than one target.

5.1.3 Drop Site

An active value whose usage is set as drop site names a target accepted by the widget as a drop site.



Usage: interface drag source [✓ drop site] widget class C++ class

Figure 5.3: Drop Site

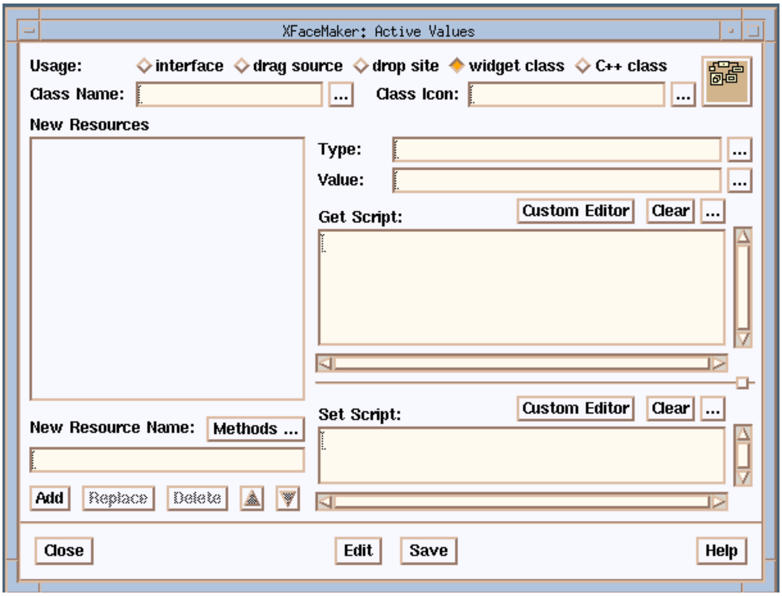
Drop Targets A display box listing the targets specified as drop sites for the selected widget.

Drop Target Name Used to specify the name of a target accepted by the selected widget as a drop site. ***List...*** Pops up a selection box of possible drop sites. Selecting one enters it in the Drop Target Name field.

The Drop Target Selector is used to select a drag-and-drop target from a list of commonly accepted selection targets as defined in the “Inter-Client Communication Convention Manual” by the X Consortium. Open the box by clicking on the ***List..*** button in the Active Values box.

5.1.4 Widget Class

An active value whose usage is set to “widget class” defines a resource for a widget class. To edit an active value of this kind, you must first select the top level widget of the class. Specify the class name and class icon--they are associated to the widget. Then define as many resources as you like for this class, and do one overall ***Save***. (You can also save your class using the commands in the Classes sub-menu of the Modules menu.)



Usage: ◇ interface ◇ drag source ◇ drop site [✓ widget class] ◇ C++ class

Figure 5.4: Widget Class

Class Name Used to specify the name of the class being defined or edited. The ellipsis opens a list of already-defined classes from which to choose. Class names must have a leading uppercase character.

New Resource Name Used to enter the resource name when you are defining a new resource, editing an existing resource, or deleting a resource.

Edit This button pops up the Class Name box from which you may select the class name that you want to edit. XFaceMaker will pop the FileSelector to let you specify the name of the .fm file that defines the class, and it will load the file.

Class Icon Enter the name of a pixmap to represent the class in the widgetstore. The ellipsis pops up the Pixmap editor for defining a pixmap.

Type Specify the active value's *type* in the "Type" field, or click the ellipsis to pop up the Type Selector box. Choose a type from the list or enter one in the "Resource type" text field, then click ***OK***.

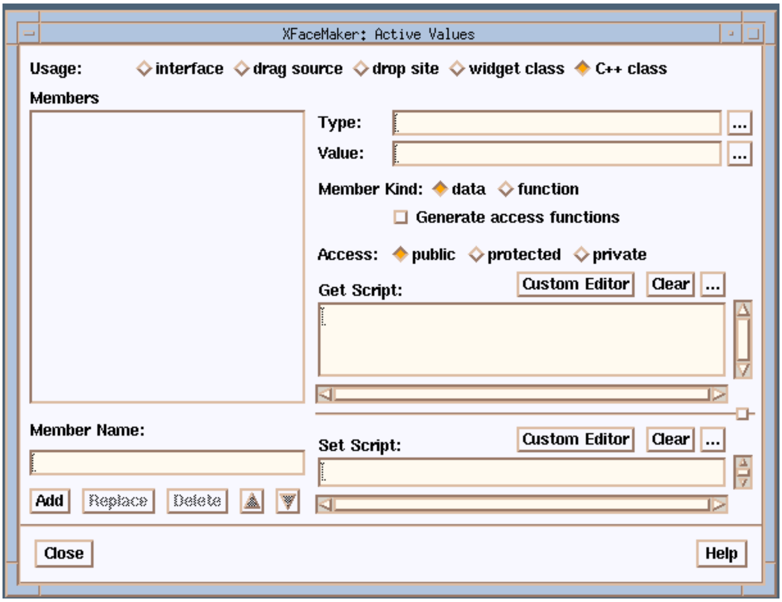
Value Enter a default value for the new resource being defined in the "Value" field. Or click on the ellipsis to open an editing window for that type of resource. Enter a value, then click on the ***Add*** button. Or click on the ***List*** button to return to the Active Values box without specifying a value.

Methods... Pops up a list of methods available for your widget class. You can select a method name from the list or enter a name directly in the text field of the Method Selector box. Then click the ***OK*** button.

See Chapter 11, New Widget Classes for details on class methods and defining resources for a widget class.

5.1.5 C++ Class

An active value whose usage is set as “C++ class” is a member of the C++ class generated in connection with the widget.



XFaceMaker: Active Values

Usage: interface drag source drop site widget class [✓ C++ class]

Members

Type:

Value:

Member Kind: [✓ data] function

☐ Generate access functions

Access: [✓ public] protected private

Get Script:

Member Name:

Set Script:

Figure 5.5: C++ Class

Member Name Enter the name of the member here.

Members This box displays existing members of the C++ class.

Type This depends on the “Member Kind”: data or function.

m **data** Enter the FACE type of the data member.

m **function** The “type” of a member function is a user-defined argument structure that defines the return type and the parameters of the function. Enter the name of the structure in the type field and define it in the set script of the function. The structure’s fields will be interpreted as follows:

- The first field corresponds to the value returned by the function. Its type (a FACE type) is the return type of the function. Its name will be used to specify the return value (unless the return type is None, which corresponds to the C++ void type).

- The remaining fields correspond to the arguments of the function. The order in which they appear in the structure will be their order in the argument list of the C++ member function. Each type is a FACE type, whose C++ counterpart will be used in the actual definition of the function.

Value This depends on the “Member Kind”: data or function.

m For a data member, enter the initial value here.

m For a member function, this field is not used and will be grayed out so that it is not editable.

Member Kind There are two choices:

m data --- to edit data members.

m function --- to edit member functions.

Generate access functions When set, specifies that XFaceMaker generate two public member functions for read and write access. The names of these functions will be built by concatenating “set” and “get” with the capitalized name of the member.

Access There are three choices of access level for the member: “public”, “protected”, and “private”. They correspond to C++ access levels.

5.2 The Alignment Box

The Alignment Box is used to align widgets relative to each other, or to a parent widget. It can also be used to apply a snap-on grid to a *composite* widget so that widgets are aligned in it as they are created, moved, or resized. Open the box by clicking on ***Alignment..*** in the Edit menu.

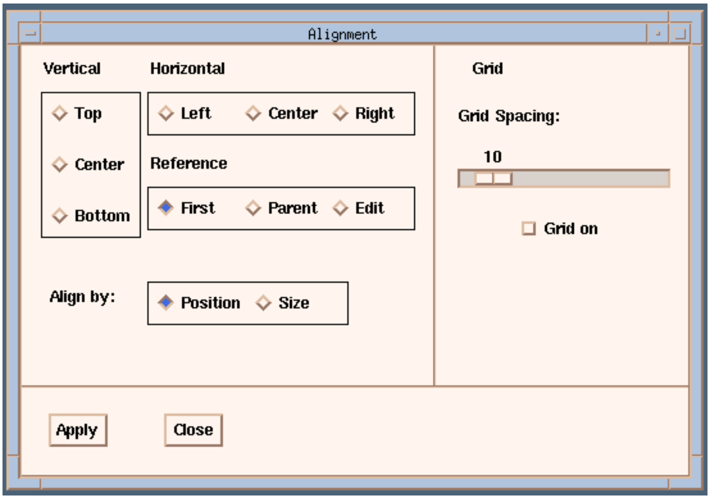


Figure 5.6: The Alignment Box

The Alignment box has the following fields:

Vertical Defines parameters for the vertical alignment.If none of the radio buttons is selected, Apply will not do vertical alignment.

Horizontal Defines parameters for the horizontal alignment. If none of the radio buttons is selected, Apply will not do horizontal alignment.

Reference Specifies the reference, i.e. the widget with respect to which a widget is to be aligned. The choices are:

- **First** – The selection will be aligned with respect to the first widget that was selected, which must be a sibling. The first widget selected has solid handles.
- **Parent** – The selection will be aligned relative to the parent widget.
- **Edit** – The selection will be aligned with respect to the last widget selected, called the *Edit* or the *current* widget, the one which has the x handles instead of the solid handles.

Align by Specifies whether the widget is to be aligned by position or size.

- **Position** – The widget will move to a new position to align as specified.
- **Size** – The widget will change size to align as specified.
- ***Grid*** The Grid field contains two items:
  - a slider used to set the spacing between lines of the grid,
  - the **On** and **Off** toggle buttons. When **On** is set, and the current widget is a BulletinBoard or a subclass of BulletinBoard, the grid is displayed; a child widget placed, moved, or resized on the composite widget will *snap* to the nearest grid line, both horizontally and vertically. When **Off** is set, no grid is displayed, widget positions and sizes are not constrained by XFaceMaker. The grid will not exist in your interface, it is only a visual and editing aid placed on the Composite by XFaceMaker.

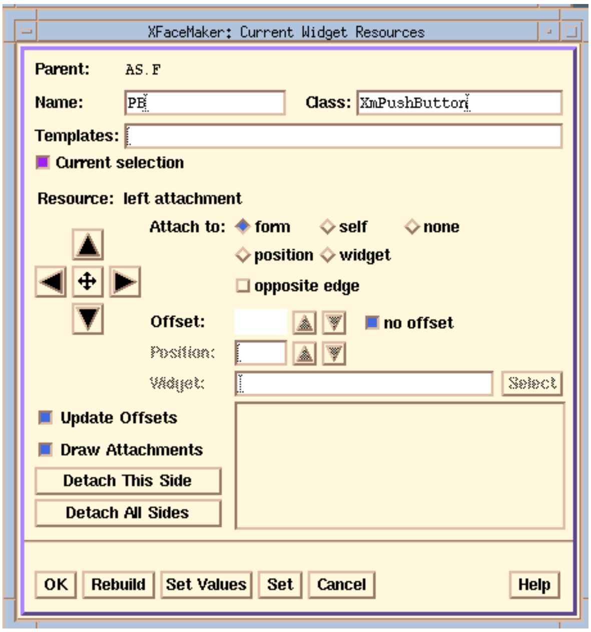
Apply Aligns the widgets as specified.

Close Closes the Alignment box. If ***Apply*** was not clicked before closing the box, no changes are made to the selected widget.

5.3 The Attachments Box

The Attachments box is used to specify widget attachments within an XmForm parent. The widget's attachments can be to its XmForm parent, to another widget, or to itself.

To view the attachments you specify, select the ***Draw Attachments*** item in the View menu. The display uses vectors originating from the selected widget edge and pointing to the reference widget's edge. Attachment vectors for **position** mode are displayed as broken lines while attachment vectors for **offset** mode are drawn with solid lines.



XFaceMaker: Current Widget Resources

Parent: AS.F

Name: PB   Class: XmPushButton

Templates:

Current selection

Resource: left attachment

Attach to: [✓ form] self none position widget

opposite edge

Offset: [✓ no offset]

Position:

Widget:

[✓ Update Offsets]

[✓ Draw Attachments]

Figure 5.7: The Attachments Box

Widgets placed within a Form composite widget inherit a set of attachment resources. The following resources define the attachment for each side of a widget:

topAttachment	bottomAttachment
leftAttachment	rightAttachment

The enumerations available are:

- attach_none do not attach current side.
- attach_form attach current side to the same side of the Form widget.
- attach_opposite_form attach current side to the opposite side of the Form widget.
- attach_widget attach current side to the corresponding side as defined in Motif (e.g., attach left side to right side), of another widget as specified in the *side*Widget resource. e.g.,

  leftWidget = parent.PushButton0
- attach_opposite_widget attach current side to the opposite side as defined by Motif, (e.g..attach left side to left side), of another widget as specified in the *side*-Widget resource.
- attach_position attach current side to a relative position within the Form widget, specifying the relative position in the *side*Position resource as a percentage of the corresponding Form's size. e.g., leftPosition = 50 attaches the left side to a point which is 50 percent of the form widgets' size (if scaleFactor is set to its default 100).
- attach_self attach current side of the widget to its original position.

The Attachments box has the following fields:

Arrow Buttons These are used to specify which border of the selected widget is to be attached: the Up arrow for TopAttachment, the Left arrow for LeftAttachment, etc.

m Once the attachment choices are set, you can apply attachments to the selected widget by clicking on one of the four arrow buttons. Move the cursor over the interface and click in the widget you want to attach to or in the Form parent.

m If "position" is selected, when you click on an arrow the selected widget is attached to its Form parent---the "?" cursor does not appear. Note that setting both an offset and a position is valid. For instance, if you want a PushButton in the middle of a Form you might give a position of 50 percent with an offset of minus half the button width.

m The cross in the middle of the arrow buttons is used to change the resource being edited. Click on it, then on the arrow corresponding to the resource (e.g. the top arrow for topAttachment).

Attach to There are five choices:

- f**orm** The widget is attached to the Form parent widget.
- **self** The widget's attachment is in respect to itself.
- **none** Specifies no attachment.
- **position** The widget is attached to the Form parent widget, with a relative position.
- **widget** The widget is attached to a sibling widget. Note that the widget you attach *to* must be created *before* the current widget.

- ***no offset checkbutton*** When set, the attachment is done without offsets; that is, the object will be attached flush to the Form, or to the sibling you specify, or at the relative position you specify.
- ***Offset*** Sensitive only if the *no offset checkbutton* is *unset*. The value displayed in this field specifies the current offset. To change this, enter a new value directly, or drag the object on the interface (if "Update Offsets" is set).

Position Sensitive only if *attach to position* has been selected.In this mode, the widget's attachment is always to the Form parent's border. If the interface is resized, a widget thus attached keeps its position relative to the Form's border. Enter the widget's position in this field.

Widget If you selected "widget" in the "Attach to" field, you must enter the name of that widget in this field. You have several ways to do so. You can enter the name directly. Or you can click on the ***Select*** button, move the cursor over the widget or widget name and click. Or you can select the widget name from the list in the field below.

Update Offsets When set, the widget's offsets will be updated when it is moved. This option is also available in the Options menu under ***Update Offsets on Move.***

Draw Attachments Specifies that widget attachments be visible when XFaceMaker is running. They will not be visible when the interface is created by the application.

Detach This Side When clicked, this button removes all attachment resources for the corresponding side of the widget currently being edited. Always remove previous attachments if you want to redefine an attachment. Otherwise, you may have stray attachment resources in your .fm file, due to the fact that, depending on the type of attachment, you choose, different resources are involved.

Detach All Sides When clicked, this button removes all attachment resources for all sides of the currently edited widget. Detach all sides of a widget if you want to remove all its attachment resources, or if you want to modify the attachments for more than one side.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the changes by rebuilding, and remains in the editor.

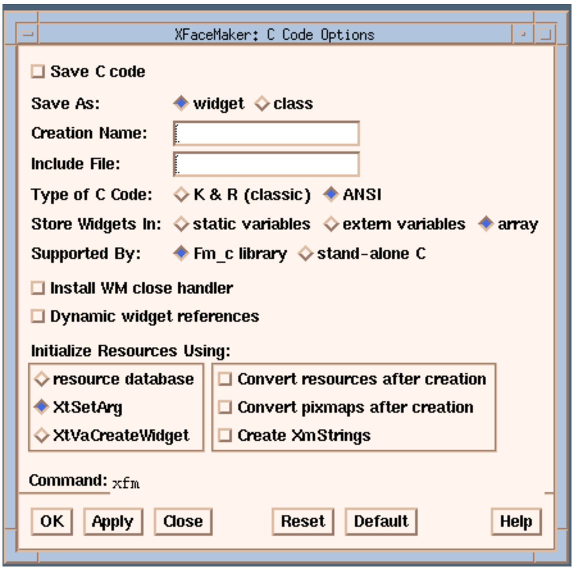
Set Values Applies the changes by setting the values and remains in the editor.

Cancel Returns to the main Resource window without changing the value.

Set Updates the value in the Resources list, but does not apply it.

5.4 C Code Options

The C Code Options box is used to set options for generating C code. All these options have a corresponding command line option.



XFaceMaker: C Code Options

☐ Save C code

Save As: [✓ widget] class

Creation Name:

Include File:

Type of C Code: K & R (classic) [✓ ANSI]

Store Widgets In: static variables extern variables [✓ array]

Supported By: [✓ Fm_c library] stand-alone C

☐ Install WM close handler

☐ Dynamic widget references

Initialize Resources Using:

resource database [✓ XtSetArg] XtVaCreateWidget

☐ Convert resources after creation

☐ Convert pixmaps after creation

☐ Create XmStrings

Command: xfm

Figure 5.8: The C Code Options Box

The C Code Options box has the following fields:

Save C code When this checkbutton is set and applied, XFaceMaker will generate a C language version of your interface whenever you save.

The following options are available:

Save As: widget

Compiles the current interface into a single creation function that can be called to create the interface, that is, FmCreate<*Name*>. The default value for *Name* is the basename of the current filename, i.e., the .fm *filename*.

Save As: class

Compiles the interface into the code necessary to create a new widget subclassed from the class of the top widget in the interface. Code is also generated to integrate the new widget class into XFaceMaker. The option Store Widgets In: array (-cflags a) will be forced to True.

Creation Name

Used to enter the name used in the creation function so that it becomes FmCreate<*Name*>. The default value for *Name* is the basename of the current filename; i.e. the .fm *filename*.

Class Prefix

Used to enter the prefix string to be used for the widget class when generating the class. The prefix is used in conjunction with the creation name to give FmCreate<*PrefixName*> which will create an instance of the new widget class <*PrefixName*>. The default value for *Prefix* is Xfm.

Include File

Used to specify an include file name to be inserted in the C-code file. The string given, e.g.”e_retApp.h” is inserted directly into the file to give the line #include “e_retApp.h”. Any special characters must be protected when used from the shell. Only one file can be included.

Type of C Code: K&R(classic)

Generates C-code conforming to the standard defined by Appendix A, of Kernigan and Ritchie’s *The C Programming Language*.

Type of C Code: ANSI

Generates C-code with prototyped functions conforming to the *ANSI C* standard. The code can also be compiled using a C++ compiler. Types defined in the application must be declared.

Store Widgets In: static variable

Specifies that when the C-code file is generated, teh ID of every resolvable widget of the interface be stored in a private C variable. For any unresolvable widget names, a dynamic widget reference is made so that the name is resolved at runtime.This option must not be used for interface fragments which will be instantiated more than once.

Store Widgets In: extern variables

Specifies that widget ID’s be stored in global (extern) C variables in the generated C file. A widget whose name is *Name* will be stored in a C variable called Fm*Name*Widget. If two widgets in the interface have the same name, one of the variables is re–named as Fm*Name*Widget_*N*, where *N* starts at 1 and is incremented for every synonym. This option must not be used for interface fragments which will be instantiated more than once.

Store Widgets In: array

Specifies that widgets be stored in an array that is allocated every time the interface creation function is called. When this option is set, XFaceMaker ignores the -compilegroup option (-cflags d), and outputs dynamic widget references only for widgets defined outside of the module. This option is general: it can be used for interface fragments that are to have multiple instances.

Supported by: Fm_c library

Generates code which makes use of the Fm_c library functions. The functions simplify the application and reduce code size. The source code of the Fm_c library is included in the standard distribution.

The following two options are valid with “Use Fm_c library”:

Install WM close handler

Installs a default handler that will be called when the Motif Window Manager ***Close*** menu item is selected, as in interpreted mode. The default handler exits the application if the shell is an ApplicationShell, or else unmaps the shell. The handler can be changed using the FmSetCloseHandler function.

Dynamic widget references

Specifies that XFaceMaker produce dynamic widget references for all widgets.

Standalone C

Specifies that the code generated contain calls to the X and Motif libraries only. The application will have to use XtInitialize and XtMainLoop instead of FmInitialize and FmLoop to be fully independent of the Fm_c library. Resource specifications in FACE scripts need to explicitly type variables. See the section on Stand-alone C code in the *XFaceMaker User's Guide*.

Initialize Resources Using: resource database

Specifies that an internal resource database be generated as an array which is passed to the X resource manager. The widget creation functions consult the resource manager which handles all resource conversions. This option has been kept for backward compatibility. It is somewhat old-fashioned.

Initialize Resources Using: XtSetArg

Tells XFaceMaker not to generate a resource database, but to call XtSetArg for all resources. The resources are either passed directly in one call, or converted from their String representation to their own representation and then passed as parameters to XtSetArg. The following resources are passed directly:

Strings

Booleans

KeySyms (call to XStringToKeysym)

integers (Int, Position, Dimension, etc.)

Widgets

enumerated constants (LabelType, Orientation, etc.)

XmStrings (optionally, see Fm*CreateXmString*)

All the other resources are converted by calling XtConvertAndStore *before* creating the widget, passing the parent widget.

If the XtSetArg option is set, you have the following options:

Convert resources after creation

Specifies that for resources that need to be converted, XtConvert be called *after* the widget has been created (passing the new widget to the converter), and set using XtSetValues.

Convert Pixmaps after creation

Specifies that for Pixmap resources that need to be converted, XtConvert be called *after* the widget has been created (passing the new widget to the converter), and set using XtSetValues. For other resources that need to be converted, XtConvert is called with the parent widget before creating the widget.

Create XmStrings

Specifies that resources of type XmString do not call the resource converter, but directly call the function *XmStringCreateLtoR*, to produce shorter code.

Initialize Resources Using: XtVaCreateWidget

Tells XFaceMaker to use the Xt "varargs" functions to create the widgets (XtVaCreateWidget, etc.). The resources of the widget are passed as a varargs list of pairs (name,value), or as a tuple (XtVaTypedArg, name, type, value, size) for resources that need to be converted.

The following three options are valid with XtVaCreateWidget:

5 Creation Arguments allowed

When set, the maximum number of arguments that can be passed to the creation function of a widget is five.

20 Creation Arguments allowed

When set, the maximum number of arguments that can be passed to the creation function is twenty.

100 Creation Arguments allowed

When set, the maximum number of arguments that can be passed to the creation function is one hundred.

The pushbuttons at the bottom of the C Code Options box do the following:

OK Applies the options and closes the window.

Apply Applies the options and remains in the C Code Options window.

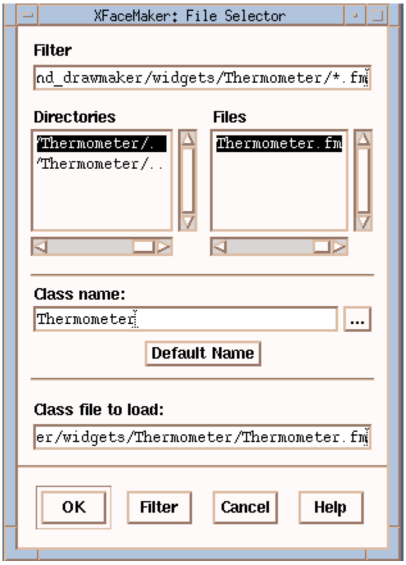
Close Closes the window without applying any options.

Reset Resets the options to the preceding values.

Default Resets the options to their default values.

5.5 The Class Name Box

Class Name box is a modified file selector used to specify the name of an Xt widget class. The box pops up automatically when you save or load a class using the commands in the Modules menu.



nd_drawmaker/widgets/Thermometer/*.fm

Thermometer/.
Thermometer/..

Thermometer.fm

Thermometer

er/widgets/Thermometer/Thermometer.fm

Figure 5.9: The Class Name Box

The Class Name Box has the following fields:

Class name Enter a name for the class here or click on the ellipsis to pop up a list of current class names.

Default Name Clicking here assigns to the class the name given in the “Class file to load field” starting with an uppercase letter.

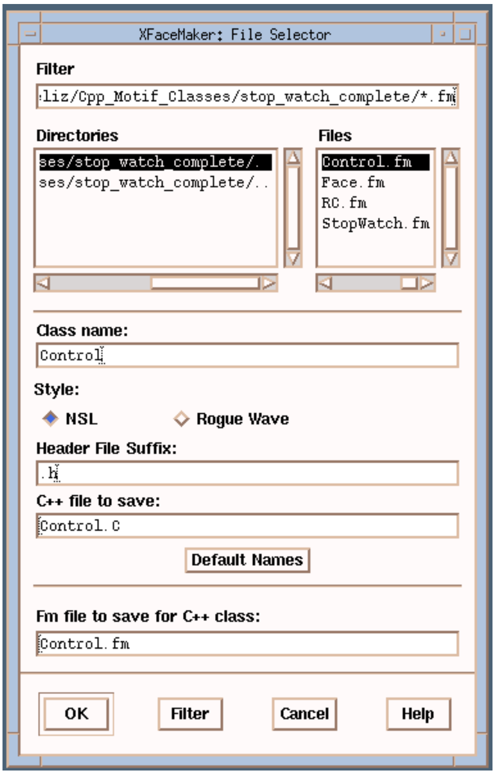
Class file to load Enter a class here or select it from the list in the “Files” field. The class name must begin with an uppercase letter.

OK Registers/selects the specified widget class, then closes the box.

Cancel Closes the Class Name box without making any changes.

5.6 C++ Class Name Box

This box is an enhanced file selector which allows you to specify the name of the C++ class, and the files in which its definition must be saved.


XFaceMaker: File Selector

Filter: :liz/Cpp_Motif_Classes/stop_watch_complete/*.fm

Directories:
```
ses/stop_watch_complete/.
ses/stop_watch_complete/..
```

Files:
```
Control.fm
Face.fm
RC.fm
StopWatch.fm
```

Class name: Control

Style: [✓ NSL] Rogue Wave

Header File Suffix: .h

C++ file to save: Control.C

Fm file to save for C++ class: Control.fm

Figure 5.10: C++ File Selector

The box has the following fields:

Class name Enter a name for the class here, or click the ***Default Names*** button once you have specified the .fm file name below.

Style Click here to choose the style used to generate the C++ files. The default is NSL style.

Header File Suffix Specify the suffix for the file that contains the C++ definition of the class.

C++ File to Save Enter the name for the C++ source file of the class or click the ***Default Names*** button once you have specified the .fm file.

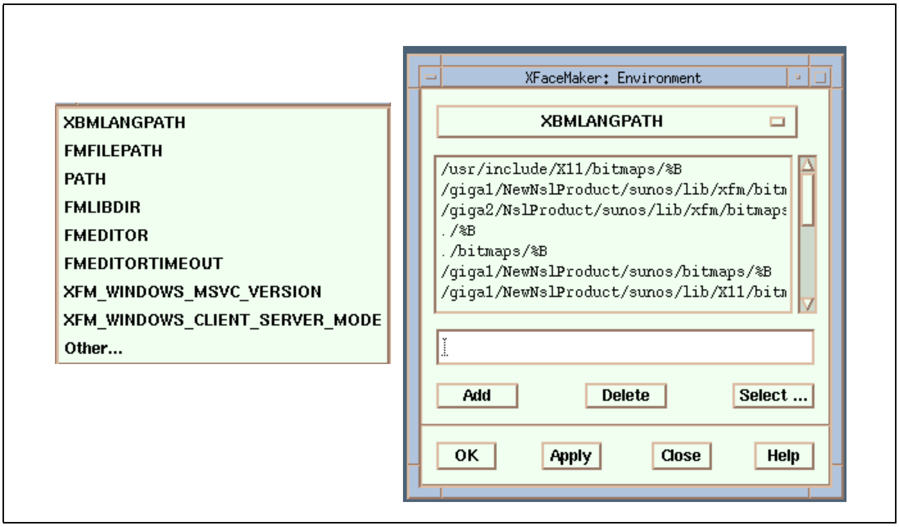
Default Name Clicking here automatically builds the class name and the C++ file name from the name of the .fm file.

Fm File to save for C++ class This field is for the name of the .fm file to save. It is the XFaceMaker model of the C++ class. This is the file you will load in XFaceMaker if you want to edit the class definition.

OK Generates the code for the specified class, then closes the box.

Cancel Closes the box without making any changes.

5.7 The Environment Box



```
/usr/include/X11/bitmaps/%B
/giga1/NewNslProduct/sunos/lib/xfm/bitn
/giga2/NslProduct/sunos/lib/xfm/bitmaps
./%B
./bitmaps/%B
/giga1/NewNslProduct/sunos/bitmaps/%B
/giga1/NewNslProduct/sunos/lib/X11/bitn
```

Figure 5.11: The Environment Box

Variable Name pulldown menu Contains a number of useful environment variables. Select an existing variable, or select ***Other...*** and type the variable name in the text field on the right (you may have to enlarge the box to display the variable name textField). If the variable is a path, the elements of the path are displayed in the scrolled list below the menu.

Add Adds the value typed in the text field to the selected path variable, after the selected line.

Delete Deletes the selected line from the path.

Select Pops up a file selector, and puts the result in the text field.

OK Stores the currently displayed value for the selected variable and closes the Environment box.

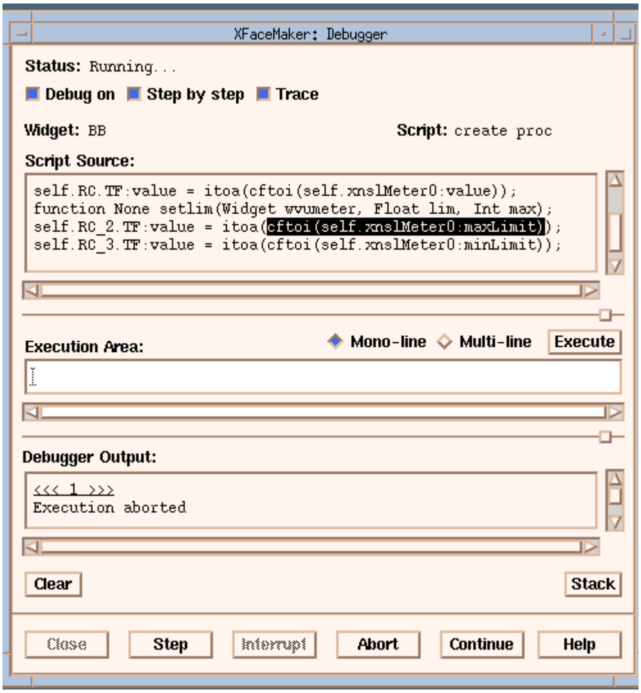
Apply Stores the value without closing the box.

Close Closes the box without modifying the variable.

5.8 The FACE Debugger

This section describes the FACE Debugger window. This window enables you to:

- trace the execution of FACE scripts.
- execute instructions step-by-step.
- print the stack of script and function calls.
- set named breakpoints.
- interrupt and abort execution of scripts.
- execute FACE statements to print variables, etc.



Status: Running...

Widget: BB

Script: create proc

Script Source:

```
self.RC.TF:value = itoa(cftoi(self.xnslMeter0:value));
function None setlim(Widget wvumeter, Float lim, Int max);
self.RC_2.TF:value = itoa(cftoi(self.xnslMeter0:maxLimit));
self.RC_3.TF:value = itoa(cftoi(self.xnslMeter0:minLimit));
```

Execution Area:

Debugger Output:

```
<<< 1 >>>
Execution aborted
```

Figure 5.12: The FACE Debugger

Test your interface by putting XFaceMaker into Try mode and executing the script; i.e., perform the action that launches the script. If an error is detected, it will be displayed in the Script source field and an error message will appear in the Debugger output field.

Errors or warnings encountered by the FACE parser or interpreter stop the debugger at the point of the error and start step-by-step mode. Stepping, aborting or continuing after an execution error terminates the Design process. The explanatory text of the error is printed in the Debugger output area as well as in the XFaceMaker message window.

You can set breakpoints in scripts by inserting calls to the ‘breakpoint’ function, which takes two arguments: a widget (usually ‘self’) and the breakpoint name (a character string). When the breakpoint is reached, the debugger enters step-by-step mode and a message containing the breakpoint name is printed in the status area.

Debugging slows down the execution of scripts. Therefore, when you are not debugging scripts, you should deselect the ***Debug On*** checkbutton.

The Face Debugger window has the following fields:

Debug On When set, this toggle button activates the FACE debugger. This button is automatically unset when the FACE debugger window is hidden.

Step by step This toggle button controls whether the FACE debugger stops before it executes each instruction or not. In step-by-step mode, you can only use the FACE debugger window and control the execution with the ‘Step’, ‘Abort’ and ‘Continue’ buttons. The rest of the XFaceMaker interface as well as your user interface is locked.

Trace When this button is set, the FACE debugger traces the execution of all instructions by showing the source of the corresponding script, and by highlighting the current instruction. If this button is not set, the debugger only traces errors, warnings and breakpoints by showing the corresponding source and instruction, and by entering step-by-step mode.Note that the FACE debugger can trace the execution of all scripts---not only widget callbacks in Try mode, but also create callbacks in Build mode or active value scripts for a user-defined widget.

Script source and ***Debugger output*** These areas are read-only. If you want to modify a script, you have to switch back to Build mode, if necessary, and edit the script in the usual XFaceMaker editing window.

Execution area You can enter instructions in the Execution area, such as debug, to print the values of variables, resources, etc. The debug function is called exactly like printf, but the output is directed to the debugger’s output area instead of the standard output. Enter the instruction, then press the ***Execute*** button. (Or press ***Return*** if ***Mono-line*** is set.)

Mono-line When this is set, you can enter a single line instruction, then press ***Return*** to execute it.

Multi-line When this toggle is set, you can type several lines of debugging instructions in the ***Execution area***, but you then have to click the ***Execute*** button, not ***Return***.

Execute This button executes the FACE statements that you have entered in the Execution area. In step-by-step mode, the FACE statements are executed as if they were part of the currently debugged script. The local variables of the debugged scripts are accessible as well as all the global variables defined so far (you do not need to use the **global** keyword before referencing global variables in this case).

Stack prints the stack of scripts and functions currently called, with the argument types and values, in the output text area.

Clear clears the debugger output area.

Close hides the debugger window (and unsets 'Tracing On').

Step advances to the next instruction in step-by-step mode.

Interrupt is used to stop the execution of a script: a SIGINT signal is sent to the design process, and the debugger switches to step-by-step mode.

Abort is used to terminate execution of the current script in step-by-step mode.

Continue is used to switch to normal run mode when in step-by-step mode. Execution continues without stopping at each instruction.

Help pops up a help text box.

5.9 The File Selector

The File Selector is used to select or specify file names when you want to open a file, (***Open...***), save a file under a new name, (***Save as...***), append a file to the current file (***Read...***), or whenever a filename is needed.

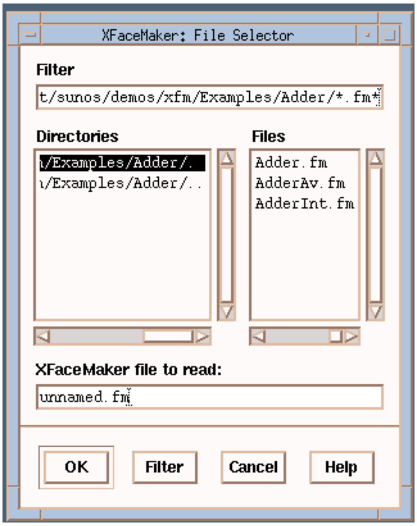


Figure 5.13: The File Selector

The File Selector box has the following fields:

Filter This text field displays and lets you edit the directory mask used to select the files to be displayed.

Directories Lists the sub-directories of the current directory.

Files Lists the files contained in the current directory. Clicking on a name in this list puts it into the Selection field.

text input field Used to edit the pathname of the current file.

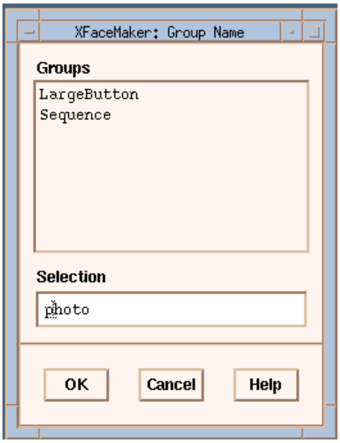
OK Reads or writes the file specified in the Selection field.

Filter Applies the directory mask and updates the sub-directories and file lists accordingly.

Cancel Closes the File Selector box without executing a command.

5.10 The Group Name Box

The Group Name Box lists the group widgets that are currently loaded in XFaceMaker. You can select one from the “Groups” list, or enter the name of a group directly in the “Group name” field. This box is popped when you *Delete* or *Load* a group.



```
LargeButton
Sequence
```

```
photo
```

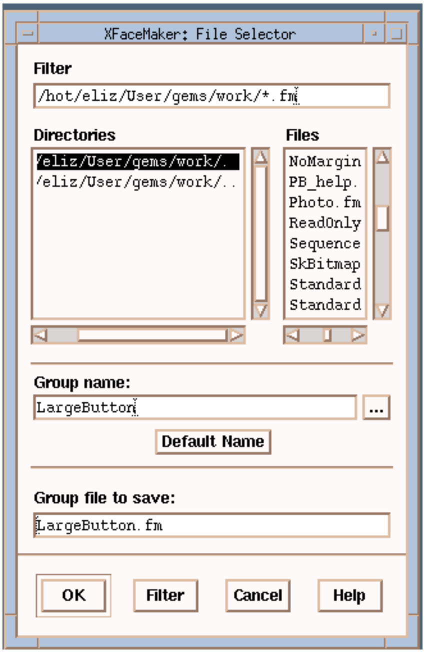
Figure 5.14: The Group Name Box

OK Click here to confirm selection of the group.

Cancel Click here to close the box without selecting a group.

5.11 The Group File Selector Box

The Group File Selector box is a modified file selector used to specify the name of a group widget when saving the selection as a group.


XFaceMaker: File Selector

Filter: /hot/eliz/User/gems/work/*.fm

Directories:
/eliz/User/gems/work/.
/eliz/User/gems/work/..

Files:
NoMargin
PB_help.
Photo.fm
ReadOnly
Sequence
SkBitmap
Standard
Standard

Group name: LargeButton

Group file to save: LargeButton.fm

Figure 5.15: The Group File Selector Box

The Group File Selector box has the following fields:

Group name This text input field is used to enter a name for the selected group. Clicking on “Default Name” will automatically enter the .fm file name here, with an uppercase first letter. The ellipsis pops up the Group Selector box.

Group file to save This text field has a default name for the group file, “unnamed.fm”. You can enter a new name here. If you click on the “Default Name” button, the name entered here will be displayed in the Group name field.

OK Registers the group name specified, and closes the Group Name box.

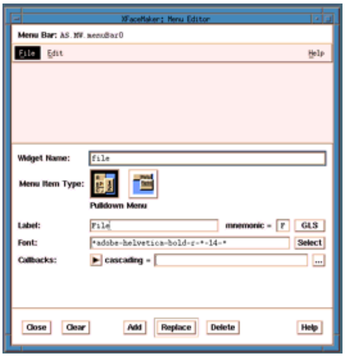
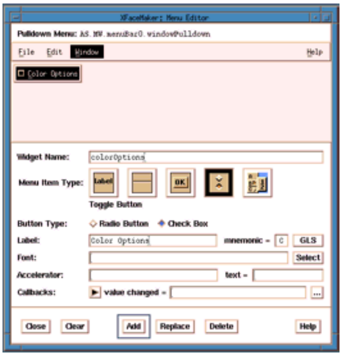
Cancel Closes the Group Name box without applying any changes.

5.12 The Menu Editor

The Menu Editor is used to build interface menus and edit their most commonly used resources. (Menu resources can also be edited in the Resources box.) To use this box, you must first select a menu icon in the toolkit's Menu widgetstore. To open this window, select ***Menu Editor*** from the Windows menu.

The Menu Editor has three sections: the Representation field, the Dialog section, and the Pushbuttons.

When you build menus, new items are added to the right or below the currently selected menu item, as appropriate. Double-clicking on a menu item opens a sub-item for you to edit.

	
Adding Cascades in a menuBar	Adding a checkBox in a pullDown

XFaceMaker; Menu Editor
Menu Bar: AS.MW.menuBar0
File Edit Help
Widget Name: file
Menu Item Type: Pulldown Menu
Label: File mnemonic = F GLS
Font: *adobe-helvetica-bold-r-*-14-* Select
Callbacks: cascading = ...
Close Clear Add Replace Delete Help

XFaceMaker; Menu Editor
Pulldown Menu: AS.MW.menuBar0.windowPulldown
File Edit Window Help
Color Options
Widget Name: colorOptions
Menu Item Type: Toggle Button
Button Type: Radio Button [✓ Check Box]
Label: Color Options mnemonic = C GLS
Font: Select
Accelerator: text =
Callbacks: value changed = ...
Close Clear Add Replace Delete Help

Figure 5.16: The Menu Editor

The Menu Editor box has the following fields, which change depending on the type of menu being edited. They are presented along with the types of widget to which they apply, if any.

5.12.1 Representation field

This field displays the menus as you build them. It works as follows:

- Each menu item is displayed with the correct label, mnemonic, and font. However, no additional graphics such as toggle button indicators, cascade indicators, or accelerators are displayed.
- The item currently being edited is highlighted, as well as the item in each menu that selects the next cascade.

- You can reorder items within a menu by dragging their menu representation, just as you would in the Widget List or Widget Tree.
- You can double–click on an item in the menu representation to open its corresponding pulldown menu. (This does not work in the Widget Tree or List, but you can still select the pulldown menu there.)

5.12.2 The Dialog Section

The dialog section of the Menu Editor is used for editing. Only the most commonly used resources are available in the Menu Editor, but you may still use the Resources box for complete control of an object's resources. The resource settings of the currently selected menu item appear in the editing fields. You can change their values, then click the ***Add*** or ***Replace*** button to apply the changes.

You can select the menus or menu items to be edited on the interface itself, in the Widget List, or in the Widget Tree. When you select an option menu, you automatically select its accompanying pulldown menu. Furthermore, you can select any item visible in the menu representation by clicking on it. You can select a pulldown menu for editing by double-clicking on its cascade button in the menu representation. The dialog section has the following fields, depending on the type of menu item being edited:

Widget Name Displays the widget name. Except for separators, you will normally not need to enter anything in it, because the menu editor automatically creates widget names based on the menu item label. If you specify the label name after the widget name, the latter is overwritten.

Menu Item Type Displays the icons of the menus or menu items available for editing. The name of the selected menu or item is displayed below the icons. These change with the type of object currently selected.

Button Type XmToggleButton This field contains two radio buttons so that you can choose the type of toggle button that will appear next to the menu item you are editing.

Label XmLabel, XmPushButton, XmToggleButton, PulldownMenu Used to enter the label seen on the menu for this item. It corresponds to the **XmNlabelString** resource. The menu editor uses this field to create a widget name, unless you explicitly enter one after specifying the label.

mnemonic Used to enter a mnemonic for the menu item being edited.

GLS Inserts the %MSG% prefix to the text entry for the label, and the mnemonic, if present.

Font XmLabel, XmPushButton, XmToggleButton, PulldownMenu Used to display or specify the name of the font used to display the menu item.

Select Opens the Fonts box for choosing the font that will be used to display the menu item being edited.

Accelerator XmPushButton, XmToggleButton Sets the key code name of the accelerator for the item being edited. Entering any keystroke in this field sets it to the key code name for that keystroke. To clear the field, clear the text field next to it.

text Displays a text description of the accelerator for the menu item being edited. This is the text appended to the button label when an accelerator is defined for the button.

Callbacks XmPushButton, XmToggleButton, PulldownMenu This field consists of three items.

- m Arrow button — Can be toggled to display the available callbacks for the selected menu item.
- m A text field — Displays the beginning of the callback specified.
- m Ellipsis button — Pops up the Script Editor box for editing callbacks.

The available callbacks are:

- **activate** XmPushButton, PulldownMenu

  The **XmNactivateCallback** resource.
- **arm** XmPushButton, XmToggleButton

  The **XmNarmCallback** resource.
- **disarm** XmPushButton, XmToggleButton

  The **XmNdisarmCallback** resource.
- **value changed** XmToggleButton

  The **XmNvalueChangedCallback** resource.
- **cascading** : PulldownMenu

  The **XmNcascadingCallback** resource.

Separator Type XmSeparator Contains a set of line types for the separator, corresponding to the **XmNseparatorType** resource. Each line has the name of the separator, and an example.

5.12.3 The Pushbuttons

The buttons on the bottom of the window provide overall control of the dialog box, and work as follows:

Close Closes the window.

Clear Replaces the values you are editing with those of the previous menu item edited.

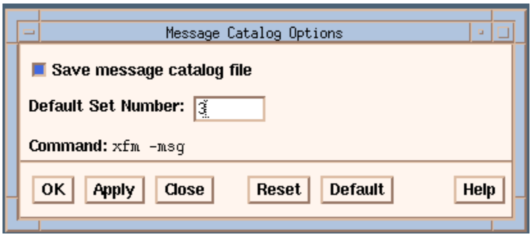
Add Adds a menu item, according to the contents of the dialog portion of the Menu Editor, to the menu shown in the menu representation.

Replace Replaces the selected menu item with a new one according to the contents of the dialog portion of the Menu Editor. Applies to the menu shown in the menu representation.

Delete Deletes the selected menu item.

5.13 Message Catalog Options Box

This box is used to specify the current message set number. The set number specified in this box is used for all %MSG% strings for which you do not explicitly specify a set number. Open it by clicking on the ***Message Catalog...*** item in the Options menu. Once you have set these options, the next time you save the interface, XFaceMaker will generate a message catalog source file (.msg file) along with the .fm file.


Message Catalog Options

Save message catalog file

Default Set Number: 3

Command: xfm -msg

Figure 5.17: The Message Catalog Options Box

The Message Catalog Options box has the following fields:

Save message catalog file When this option is set and applied, XFaceMaker generates one or several catalog files (with the .msg suffix), containing all the internationalized messages of the current interface when you save an interface.

Default set number Used to specify the current message set number for internationalized messages. When you internationalize a string without specifying the catalog set number, XFaceMaker will enter this default set number for you, and the message will be stored in that catalog.Set numbers greater than or equal to 200 are reserved.

Command Displays the current command line.

OK Applies the options and closes the box.

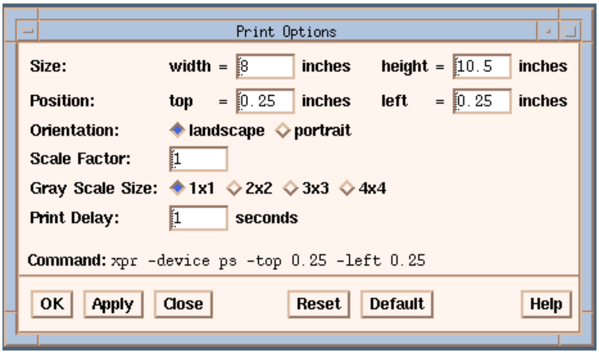
Apply Applies the options and remains in the box.

Reset Returns the box to the previous values.

Default Returns the box to the default values.

5.14 The Print Options Box

The Print Options box is used to create a postscript file for printing your interface. Open it by clicking on ***Print...*** in the Options menu.



Size: width = 8 inches height = 10.5 inches

Position: top = 0.25 inches left = 0.25 inches

Orientation: [✓ landscape] portrait

Scale Factor: 1

Gray Scale Size: [✓ 1x1] 2x2 3x3 4x4

Print Delay: 1 seconds

Command: `xpr -device ps -top 0.25 -left 0.25`

Figure 5.18: The Print Options Box

You can set the following options by entering values, and clicking the ***Apply*** button:

width The page width in inches; default value 8.

height The page height in inches; default value 10.5.

top The page top offset in inches; default value 0.

left The page left–hand side offset in inches; default value 0.

landscape Prints the window in landscape mode; default. “Landscape” orientation is a page that is wider than it is tall.

portrait Prints the window in portrait mode. “Portrait” orientation is a page that is taller than it is wide.

Scale factor Affects the size of the window on the page; the default scale is the largest that will fit on the page defined. Value can be between one and six.

Gray Scale Size Gray scale conversion of color image. Value can be between one and four.

Print delay Number of seconds to wait before storing the window; default one second.

Command Displays the specified xpr command parameters.

The pushbuttons at the bottom of the window are used as follows:

OK Sets the print options for future printing and closes the box.

Apply Applies the print options you have entered in the window to any future ***Print*** commands, but does not close the Print Options box.

Close Closes the window without applying the options.

Reset Resets the fields of the window to the previous values; i.e. the values specified at the last ***Apply***.

Default Resets all the fields of the box to their default values.

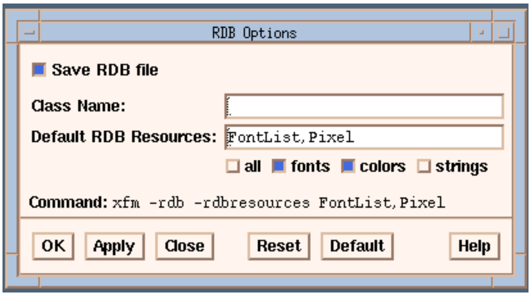
Help Pops up a Help box on using the Print Options window.

Once the options are set and applied, select the ***Save Prefs*** item in the File menu to save the print options for subsequent XFaceMaker sessions.

5.15 RDB Options

The RDB Options box is used to set the options for saving an RDB file. Whenever you save an interface with the ***Save RDB file*** option set, two files are saved in addition to the standard .fm file:

- An .rdb file, the Resource Data Base File. This contains the resource specifications for the resources selected in the RDB Options Box.
- An .fm_rdb file. This contains the interface description file in .fm format, containing the full interface description except for the resource specifications which are in the .rdb file. C code and UIL files are generated from this file if those save options have been set.



☑ Save RDB file

Class Name:

Default RDB Resources: FontList,Pixel

☐ all ☑ fonts ☑ colors ☐ strings

Command: `xfm -rdb -rdbresources FontList,Pixel`

Figure 5.19: The RDB Options Box

The following options are available:

Save RDB file When this checkbutton is set and applied, XFaceMaker will save the .rdb versions of the interface file when you save.

Class name This specifies the first component of the resource specification. Thus the name MyAppli will produce a resource file containing lines of the form:

MyAppli.name1.name2...namen: value

This allows multiple .rdb files to be generated when an interface has been split into several .fm files. Each RDB file can be given the same class name so that they can all be concatenated together to produce one application RDB file. If the class name is not defined then the name of the .rdb will be used.

Default RDB resources Used to enter other resources to be saved in the RDB file. In addition to colors and fonts, you can enter any other resource types by defining the resource by its Class (with XmC removed), Type, or its name directly, e.g., Foreground, mnemonic, Pixel, labelString. The list is comma separated without intervening spaces.

options These three checkbuttons specify the following:

all This generates an RDB file with the extension .rdb containing all the interface resources, as well as a version of the .fm file with no resources, whose extension is .fm_rdb.

fonts This generates an RDB file with the extension .rdb containing all FontList resources.as well as a version of the .fm file with no font resources, whose extension is .fm_rdb.

colors This generates an RDB file with the extension .rdb containing all Pixel, i.e. color resources, in the .rdb file.as well as a version of the .fm file with no color resources, whose extension is .fm_rdb.

strings This generates an RDB file with the extension .rdb containing all XmString resources, as well as a version of the .fm file with no compound string resources, whose extension is .fm_rdb.

Command Displays the current command line.

OK Applies the specified options when the interface is saved.

Apply Applies the options and remains in the RDB Options box.

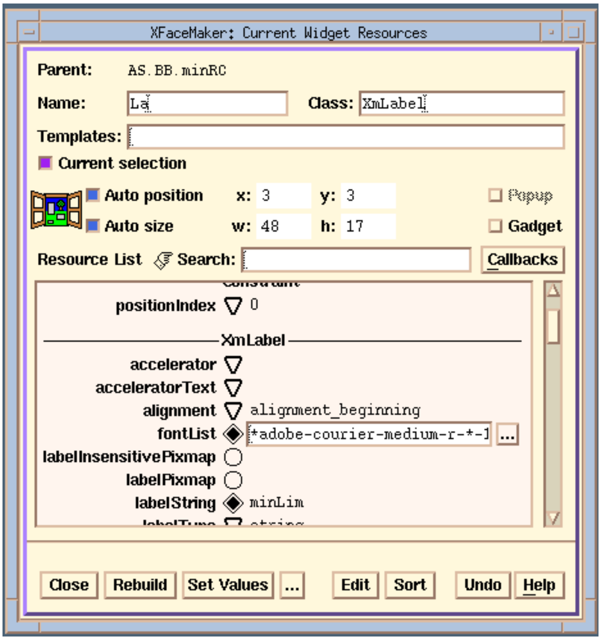
Close Closes the window without changing the RDB options.

Reset Returns the window to its previous values.

Default Returns the window to the default values.

5.16 The Resources Box

The Resources Box lists resources for the currently selected widget and enables you to edit and apply them. Changes to the selected widget are applied by pressing the ***Rebuild*** or the ***Set Values*** button, or **Return**. Open the Resources box by clicking on ***Resources*** in the Windows menu, or by clicking mouse Button 2 on a widget or its name.


XFaceMaker: Current Widget Resources

Parent: AS.BB.minRC

Name: La Class: XmLabel

Templates:

Current selection

Auto position x: 3 y: 3 Popup

Auto size w: 48 h: 17 Gadget

Resource List Search: Callbacks

positionIndex 0

XmLabel

accelerator

acceleratorText

alignment alignment_beginning

fontList *adobe-courier-medium-r-*-1 ...

labelInsensitivePixmap

labelPixmap

labelString minLim

Figure 5.20: The Resources Box

There are three sections in the Resources Box. They are:

- the Identification section.
- the ResourceslList.
- The Apply options.

5.16.1 The Identification Section

The Resources window has the following fields:

Parent Displays the name of the parent of the widget being edited.

Name Displays the name of the widget being edited. You can change the widget's name by entering a new name in this field. XFaceMaker assigns a default name to every widget in your interface, and that default name is displayed in this field until you change it. The widget name is not a resource, but is used within the interface and the application to identify and access the resources of the widget. Any legal C language variable name is a legal widget name. XFaceMaker ensures that two sibling widgets do not have the same name by adding a numeric extension to a name if it already exists, unless you use the command line option -nonunique-names. If you change the name of your ApplicationShell, the change will be displayed in the "Name" field, the Widget Tree, and the Widget List. However, the name of the ApplicationShell itself, as it appears in the title bar, is always that of the *current* application, i.e. xfm, unless you set the title resource to the new name.

Class Identifies the widget's class. This is the same as the widget type in the Toolkit, with the prefix Xm for Motif widgets.This can be edited. You can drag an icon from the Toolkit to this field to change the class.

Templates Identifies the template associated with the widget, if one is defined. You can apply or delete a template by entering or deleting its name here and clicking ***Rebuild*** or ***Set Values***. There can be more than one template listed here, separated by commas. You can also apply a template using the ***Apply/Unapply template*** commands in the Edit menu,

Current Selection Indicates if the widget is the current selection. Deselect this checkbutton to continue editing the widget even after deselecting it.

Auto position When set and applied to a widget, XFaceMaker will not specify x and y settings. Any previous settings for the x and y resources in the Resource Editor are removed. You could, for example, use this option for a button whose position is defined by its attachment resources and whose x and y resources are therefore not required. This also reduces the size of the .fm or C-code file generated. *Auto Pos* and *Auto Size* may be applied to several widgets at once, or to the whole interface, by using the menu item ***Auto Pos/Size*** in the Edit menu.

x and y Indicate the position of the upper left corner of the widget relative to the composite or shell widget that contains it; or, if it is a shell widget, to the upper left corner of the display.

The x axis increases from left to right, and the y axis downwards. All measurements are in pixels. The previously described *Auto position* option may override these fields. There is no "3-d" shadowing in these fields when "Auto pos" is set.

Auto size When this checkbutton is set and applied, XFaceMaker will not specify w and h settings.

For example, you might use this option for a label that should default to the size of its labelString resource, or for an XmList contained within an XmScrolled Window. Any previous settings for the width and height resources in the Resource Editor box are removed when Auto Size is set. This also reduces the size of the .fm or C-code file generated. Note that *Auto Pos* and *Auto Size* may be applied to several widgets at once, or to the whole interface, by using the *Edit* menu item *Auto Pos/Size*.

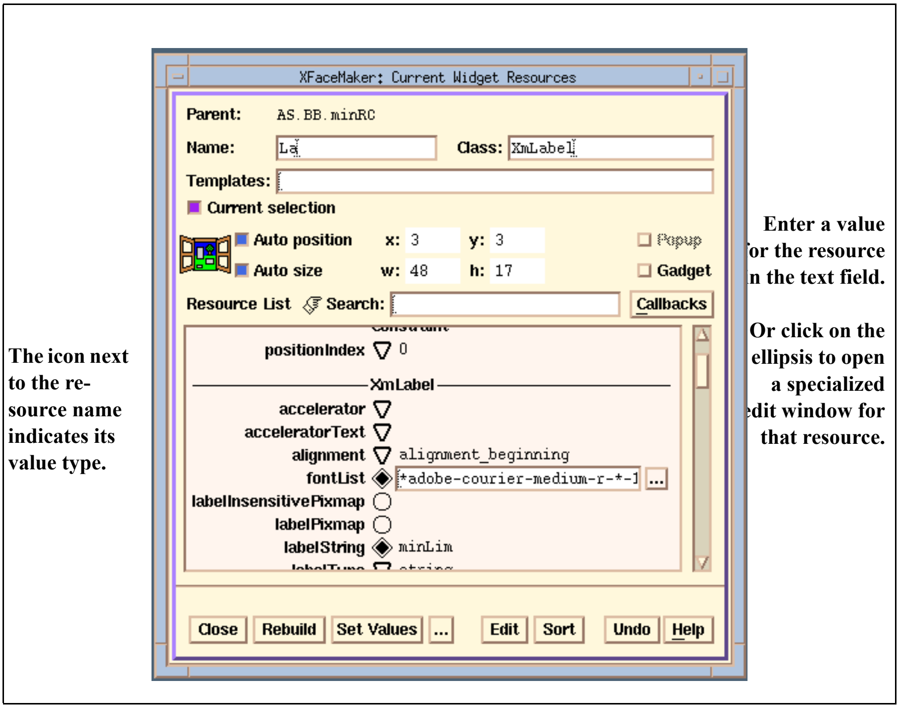
w and h Indicate the size of the current widget in pixels. The *w* parameter gives the width of a widget along the x axis and the h parameter its height along the y axis. The previously described Auto size option may override these fields. Measurements always reflect the actual values on the screen. Thus if you select a widget and move or resize it with the mouse, these values are updated automatically. You can also type in new values and apply them to the widget. Remember that monitors vary in size. Common monitor dimensions are about 1200 pixels in width and 1000 in height. A Sun 19-inch monitor is 1152 by 900 pixels, while a PC screen may only have a resolution of 1024 by 768pixels. There is no “3-d” shadowing here when “Auto size” is set.

Popup Indicates if the widget is a PopupShell. In most interfaces, any Shell widget which is not the interface’s ApplicationShell is a PopupShell and XFaceMaker will set this flag by default. Because the Xt Toolkit allows the creation of toplevel shells that are not Popup, this possibility has been implemented in XFaceMaker. However, we recommend that this flag is never changed.

Gadget When this is set and applied, XFaceMaker uses the equivalent gadget version of the current widget, if available. Some widget resources are not accessible to gadgets. XFaceMaker refuses to change a widget which uses such a resource to a gadget. Therefore, you must not use such resources for any widget you wish to change to a gadget.

5.16.2 The Resource List

The Resources List displays the resources of the widget you are editing, with an icon to indicate its value type. You can open a text field to edit the resource. The resources in the list can be sorted in various ways; for example, so that the ones you use frequently are displayed first.



The icon next to the re-source name indicates its value type.

Enter a value or the resource n the text field.

Or click on the ellipsis to open a specialized dit window for that resource.

XFaceMaker: Current Widget Resources

Parent: AS.BB.minRC

Name: La Class: XmLabel

Templates:

Current selection

Auto position x: 3 y: 3 Popup

Auto size w: 48 h: 17 Gadget

Resource List Search: Callbacks

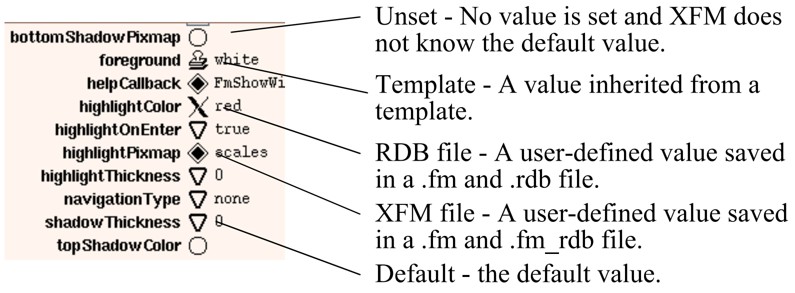
```
        positionIndex ▽ 0
─────────────── XmLabel ───────────────
          accelerator ▽
      acceleratorText ▽
            alignment ▽ alignment_beginning
             fontList ◆ *adobe-courier-medium-r-*-] ...
labelInsensitivePixmap ○
          labelPixmap ○
          labelString ◆ minLim
```

Close | Rebuild | Set Values | ... | Edit | Sort | Undo | Help

Figure 5.21: The Resource List

- To edit a resource, click on the resource line. (Clicking on the resource name also selects the resource. The resource need not be selected in order to be edited.) This opens a text input field for entering a value. The ellipsis button opens a specialized editing window for that resource. To edit callback resource, click on the Callbacks button.
- To search for a resource, scroll the list or enter its name (or the first few letters of its name, in the “Search” field and press Return. Or place the pointer over the list of resources and press the first letter of the resource’s name. The list will advance to the first resource beginning with that letter.

The icons in the Resource list indicate the following:



```
bottomShadowPixmap
        foreground   white
      helpCallback   FmShowWi
    highlightColor   red
   highlightOnEnter  true
   highlightPixmap   scales
highlightThickness   0
    navigationType   none
   shadowThickness   0
    topShadowColor
```

Unset - No value is set and XFM does not know the default value.

Template - A value inherited from a template.

RDB file - A user-defined value saved in a .fm and .rdb file.

XFM file - A user-defined value saved in a .fm and .fm_rdb file.

Default - the default value.

Figure 5.22: The Resource List Icons

You can drag and drop widgets and resource values using mouse Button 2. You may do the following operations:

- ***Drag a widget to a Resources window for editing***.

  Press mouse Button 2 on the widget, or on its name in the Widget Tree or List. The cursor changes to a drag icon. Drag the icon to a Widget Resources window, and drop it anywhere in the window.

- ***Drag a resource value to a widget.***

  Press mouse Button 2 on a text field in the list. Drag the icon over a widget or its name in the Tree or List and release. If the widget has a resource of that type, the value will be applied. If not, of course, no change is made, and an error message will appear. For instance, you cannot set an arrowDirection resource value for a PushButton.

- ***Drag a value from one resource to another resource in the list.***

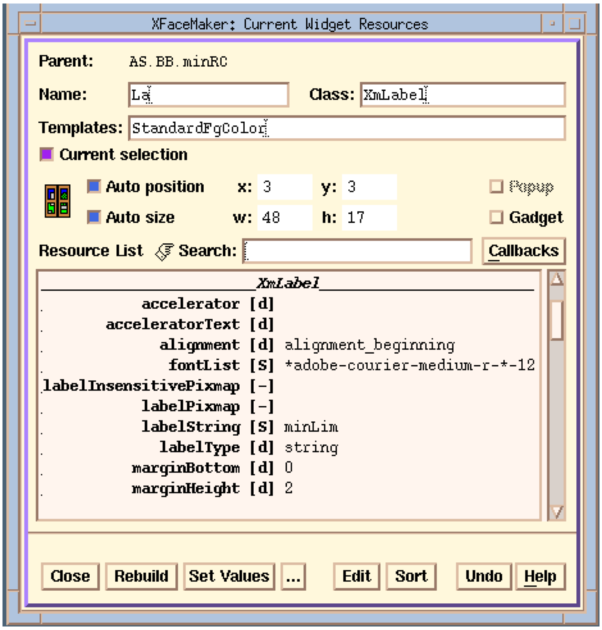
  Click to open the text field next to a resource. Press mouse Button 2 on the value listed. Drag the icon over another resource and release. You must then apply the new value by clicking ***Rebuild*** or ***Set Values***.

- ***Drag a list of selected resources to a widget.***

  Select the resources using the standard Motif extended selection process, then drag them to the widget using Button 2. This sets the resources for that widget. If you drag to a resource window, it updates the window, but you must apply the change. If you drag to a widget (or its name in the Widget List or Tree), it applies to the object, (using ***Rebuild*** on ***Set Values***) on the window where the list came from.

5.16.3 The “Fast” Resource List

This version of the Resource list is displayed if you launched XFaceMaker with the -fastlist command line option, or if you set the ***Option/Use Fast Resource List*** item.



XFaceMaker: Current Widget Resources

Parent: AS.BB.minRC

Name: La

Class: XmLabel

Templates: StandardFgColor

Current selection

Auto position x: 3 y: 3

Auto size w: 48 h: 17

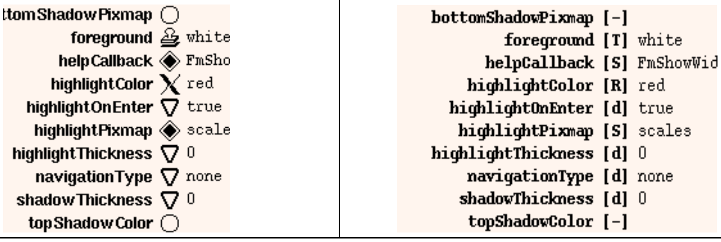
Popup

Gadget

Resource List Search:

```
                         XmLabel
             accelerator [d]
         acceleratorText [d]
               alignment [d] alignment_beginning
                fontList [S] *adobe-courier-medium-r-*-12
labelInsensitivePixmap [-]
             labelPixmap [-]
             labelString [S] minLim
               labelType [d] string
            marginBottom [d] 0
            marginHeight [d] 2
```

Figure 5.23: The “Fast” Resource List



```
ttomShadowPixmap ○          bottomShadowPixmap [-]
      foreground ⚖ white            foreground [T] white
    helpCallback ◆ FmSho          helpCallback [S] FmShowWid
  highlightColor ✗ red          highlightColor [R] red
highlightOnEnter ▽ true       highlightOnEnter [d] true
 highlightPixmap ◆ scale       highlightPixmap [S] scales
highlightThickness ▽ 0      highlightThickness [d] 0
  navigationType ▽ none         navigationType [d] none
 shadowThickness ▽ 0           shadowThickness [d] 0
  topShadowColor ○              topShadowColor [-]
```

Figure 5.24: The Fast List Letters

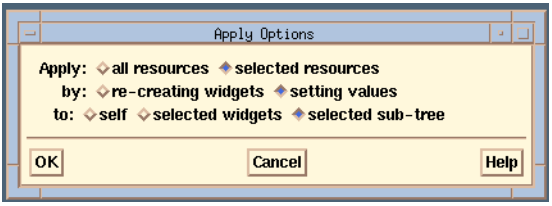
The figure above shows the letter/icon correspondence.

5.16.4 The Apply Options

Once you have specified new resource values for the widget you are editing, apply them to the widget by clicking ***Rebuild*** or ***Set Values.*** The default options are:

- all resources
- re-creating widgets
- self

To override the default options for this apply, set new values in the Apply Options box and click ***OK***. Open the Apply Options box by clicking on the ellipsis next to the ***Set Values*** button.



Apply Options

Apply: all resources [✓ selected resources]
by: re-creating widgets [✓ setting values]
to: self selected widgets [✓ selected sub-tree]

Figure 5.25: The Apply Options

You can apply:

- **all resources** -- Applies all changes made since the last ***Rebuild*** or ***Set Values***. This is the default.
- **selected resources** -- Applies only the selected resources in the list.

The new resource values can be applied by:

- **re-creating widgets** -- A new instance of each object implied in the resource change will be created for each resource modification. The effects of the change are immediately visible. This is the default mode.
- **setting values** -- Choosing this option applies the new resource values to existing objects. Most resource changes are directly visible, although some are not. Some resources cannot be changed this way, and an error message will be generated.

The new resource values can be applied to:

- **self** -- Applies the values to the widget being edited in that window. This is the default.
- **selected widgets** -- Applies the values to all selected widgets.
- **sub-tree** -- Applies the values to the current edit widget and all of its descendants.

You can undo the last change made to the resources of the widget you are editing, by clicking on the ***Undo*** button at the bottom of the resources window. Doing so replaces the change with the previous values. If there is nothing to undo, a message stating so is output in the message area of the main window. Each time you click on *Undo* you will undo an earlier change until no more can be undone.

Close Closes the Resources window.

Rebuild Applies the resources by rebuilding, and remains in the editor.

Set Values Applies the resources by setting values, and remains in the editor.

...(ellipsis) Pops up the Apply Options box.

Edit Changes to the Resource Edit window. The button’s lable changes to List, indicating that clicking on it will return the window to the resources list.

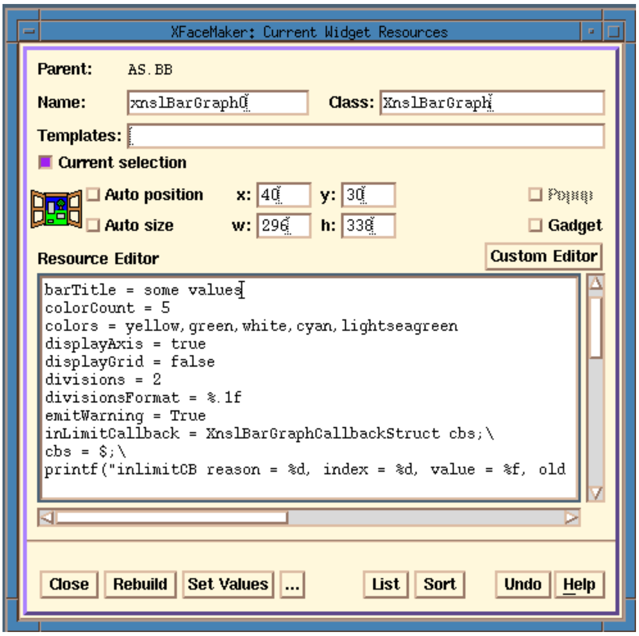
Sort Changes to the Sort Order window.

Undo Undoes the last resource modification.

.

5.16.5 The Resource Editing Window

The Resource Editor box is used to edit resources for a selected widget. Open it by clicking on ***Edit*** or *Callbacks* in the Resources box.



Parent: AS.BB

Name: xnslBarGraph0 Class: XnslBarGraph

Templates:

x: 40 y: 30

w: 296 h: 338

```
barTitle = some values
colorCount = 5
colors = yellow, green, white, cyan, lightseagreen
displayAxis = true
displayGrid = false
divisions = 2
divisionsFormat = %.1f
emitWarning = True
inLimitCallback = XnslBarGraphCallbackStruct cbs;\
cbs = $;\
printf("inlimitCB reason = %d, index = %d, value = %f, old
```

Figure 5.26: The Resource Editing Window

The Resource Editing window has the following fields:

A display window that displays resources and callbacks. If you open it with the ***Edit*** button, it displays all the resources and callbacks that have been specified for the selected widget. You can edit resources directly in this window. If you open it for a ***callback***, it displays and lets you edit only the callback for which it has been opened. In both cases, you can open the custom editor to do your editing in your favorite editor, rather than in the (default) scrolledText.

Close Only if the resource editor was opened with the ***Edit*** button. Closes the Resources window.

Rebuild Applies the specified resources by rebuilding.

Set Values Applies the resources by setting values.

List Only if the resource editor was opened with the ***Edit*** button. Changes back to the Resource List.

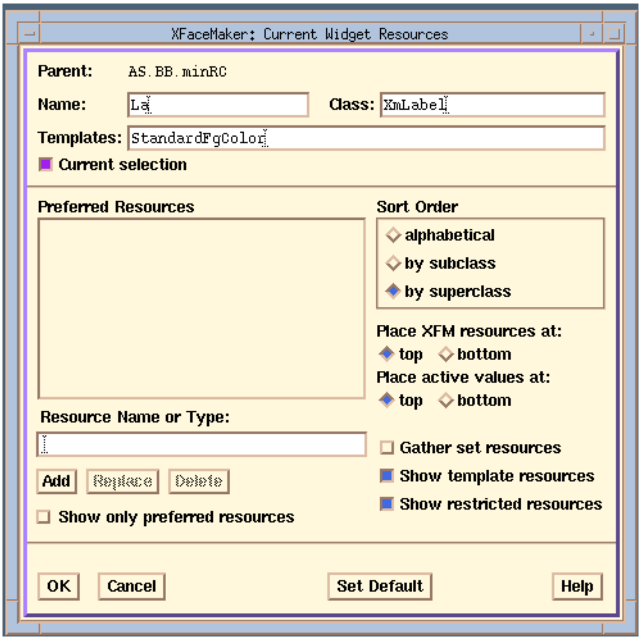
Sort Only if the resource editor was opened with the ***Edit*** button. Changes to the Sort Order window.

Undo Only if the resource editor was opened with the ***Edit*** button. Undoes the last resource modification.

Select Only if the resource editor was opened to edit a specific callback. Clicking on the button changes the cursor to the “?” cursor with which you can select the widget that you want to access within your callback. The relative path to the widget will be pasted directly in your editing window when you click the “?” on the desired widget.

5.16.6 The Sort Order Window

This window provides options for sorting and displaying resources in the Resources list. Sort options are saved when you select the ***Save Prefs*** command in the File menu.


XFaceMaker: Current Widget Resources

Parent: AS.BB.minRC

Name: La

Class: XmLabel

Templates: StandardFgColor

[✓ Current selection]

Preferred Resources

Resource Name or Type:

Show only preferred resources

Sort Order

alphabetical

by subclass

[✓ by superclass]

Place XFM resources at: [✓ top] bottom

Place active values at: [✓ top] bottom

Gather set resources

[✓ Show template resources]

[✓ Show restricted resources]

Figure 5.27: The Sort Order Window

The Sort Order window has the following fields:

Preferred Resources This field displays the list of preferred resources that you have chosen for the widget you are editing. The list is established by entering the resource name in the "Resource Name or Type" field, and clicking ***Add***. You can also delete or replace resources in this list.

Show only preferred resources When this checkbutton is set and applied, only preferred resources will be displayed in the Resources list.

Resource Name or Type A text input field for entering a resource name to be added or deleted from the Preferred Resources list.

Add Clicking here adds the resource specified in the "Resource Name or Type" field to the list of preferred resources.

Replace Clicking here replaces the selected resource in the list of preferred resources with the resource specified in the “Resource Name or Type” field.

Delete Clicking here deletes the selected resource from the list of preferred resourc-es.

Sort Order This field gives you options for changing the sort order and display op-tions of the Resources List. The sort order options are:

- **alphabetical** -- Present resources in alphabetical order.
- **by subclass** -- Group resources by subclass;.i.e. the widget-specific resources are first, followed by inherited, then core resources.
- **by superclass** -- Group resources by superclass i.e., the core resources come first, followed by any inherited resources, and ending with the current wid-get’s resources.

Place XFM resources at: top, bottom Displays the XFM resources at the top or bot-tom of the list, as specified.

Place active values at: top, bottom Displays the active values at the top or bottom of the Resources list, as specified.

Gather set resources Setting this option groups resources that have user-defined values, and displays them at the top of the Resources list, or after the “Preferred Resources” if you have established that list.

Show template resources -- Setting this option displays any template resources that have been set.

Show restricted resources -- Setting this option displays any restricted resources. These are resources that cannot be changed in XFaceMaker, such as colormap, numChildren, screen, or template resources.

OK Applies the choices and re-opens the Resource List.

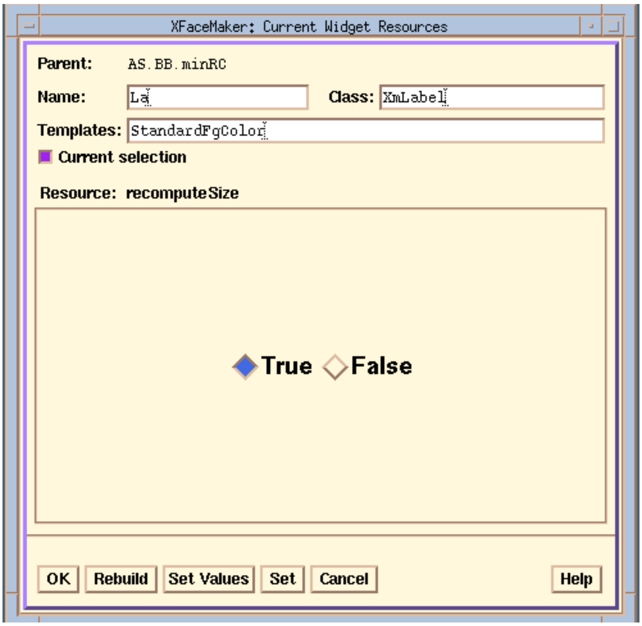
Cancel Returns to the Resource List without applying a choice.

Set Default Make your sort and display choices the default case for any *new* Re-source windows you open by clicking on this button.

5.17 The Resource Edit Boxes

5.17.1 The Boolean Box

The Boolean box is used to set a resource value to **True** or **False**. When you are editing a resource that requires a boolean value, you can open this box by clicking on the ellipsis next to the resource.



Parent: AS.BB.minRC
Name: La
Class: XmLabel
Templates: StandardFgColor
■ Current selection
Resource: recomputeSize
◆ True ◇ False

Figure 5.28: The Boolean Box

The Boolean box has the following fields:

True/False Used to specify the resource value.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the changes by rebuilding, and remains in the editor.

Set Values Applies the changes by setting values, and remains in the editor.

Cancel Returns to the Resource window without changing the value.

Set Updates the value in the Resources list, but does not apply it.

5.17.2 The Colors Box - Color Display

The Colors box, shown below as it would appear on a color display, is used to select a color for the widget resource being edited. It is popped up when color or pixel resources for a widget are being edited.

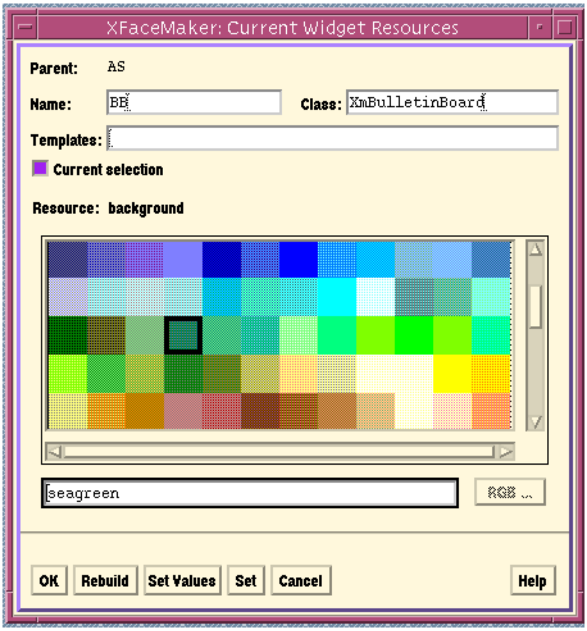


Figure 5.29: The Color Box---color

It has the following fields:

color palette Clicking on a color square selects the color to be used for the selected resource, and its name appears in the text field. The resource name is displayed above the color palette.

text input line Displays the name of the selected color, or can be used to enter the name or pixel value directly.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the changes by rebuilding the widget, and remains in the editor.

Set Values Applies the changes by setting the values and remains in the editor.

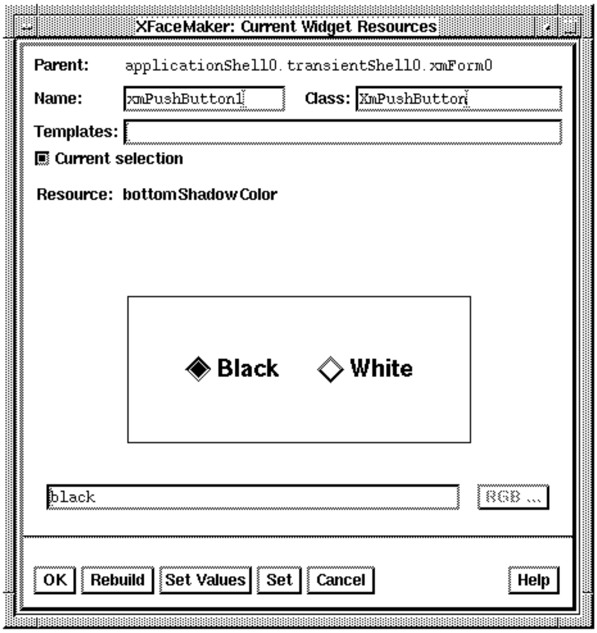
Cancel Returns to the Resource window without changing the value.

Set Updates the value in the Resources list, but does not apply it.

RGB... Pops up the RGB color editing box.

5.17.3 The Colors Box - Monochrome Display

The Colors box, shown below as it would appear on a monochrome display, is used to select a color for the widget resource being edited. It is popped up when color or pixel resources for a widget are being edited.



Parent: applicationShell0.transientShell0.xmForm0
Name: xmPushButton1
Class: XmPushButton
Templates:
Current selection
Resource: bottomShadowColor
◆ Black ◇ White
black

Figure 5.30: The Color Box---monochrome

It has the following fields:

Black White Specifies the color to be used for the resource whose name is displayed above the color values.

text input line Displays the name of the selected color, or can be used to enter the name or pixel value directly.

OK Registers and applies the selected color to the widget resource being edited.

Rebuild Applies the changes by rebuilding, and remains in the editor.

Set Values Applies the changes by setting the values, and remains in the editor.

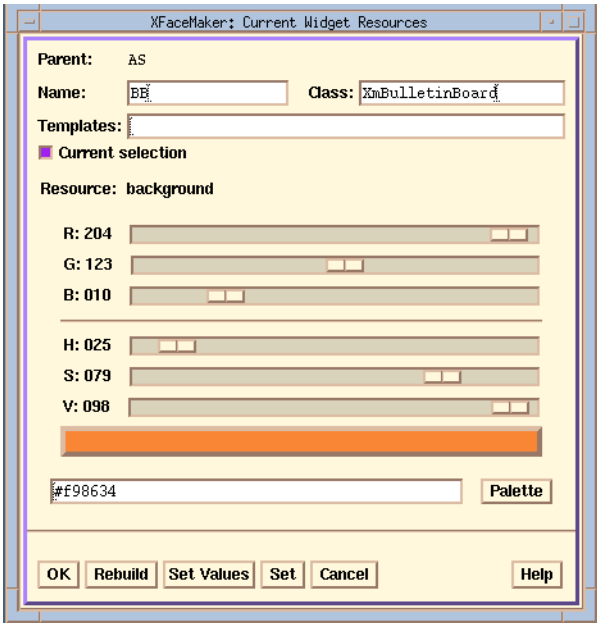
Cancel Closes the Colors box without registering a color for the selected widget resource.

Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the main Resource window without changing the value.

5.17.4 The Colors Box - RGB Box

The RGB box is used to specify the RGB (Red, Green, Blue) value or HSV (Hue, Saturation, Value) value for a color resource. RGB and HSV definition are only available if you use XFaceMaker on a color display with a read/write color table, i.e., with PseudoColor or DirectColor visuals. Open it by clicking on the ***RGB...*** button in the Colors box.


XFaceMaker: Current Widget Resources

Parent: AS

Name: BB

Class: XmBulletinBoard

Templates:

Current selection

Resource: background

R: 204

G: 123

B: 010

H: 025

S: 079

V: 098

#f98634

Figure 5.31: The RGB Editor

Use the sliders to set the values. The actual color is shown in a pad below, and its hexadecimal representation is listed in the text field. The text field may also be edited directly.

The box has the following sliders:

Red/Green/Blue These sliders enable you to define a new color in the RGB representation.

Hue/Saturation/Value These sliders enable you to define a new color in the HSV representation.

display pad Displays a sample of the color defined in the RGB box, fetched from the

X server.

text input line Displays the name or hex value of the color defined in the RGB box; it can be used to enter a color name directly.

OK This button applies the specified color values and returns to the main Resource window. If you have defined a new color, the file selector will pop up so that you can save its definition in the color description file.

Rebuild Applies the selected values by rebuilding the object and remains in the RGB box.

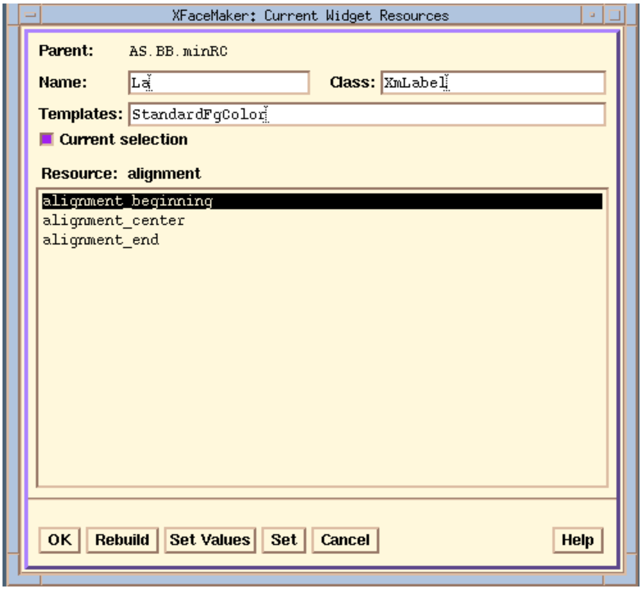
Set Values Applies the selected values by setting the values and remains in the RGB box.

Cancel Returns to the main Resource window without changing the value.

Set Updates the value in the Resources list, but does not apply it.

5.17.5 The Enumeration Box

The Enumeration box displays a list of possible resource values, with the current setting highlighted. This box is popped up when you select an enumerated resource in the Resources box.



Parent: AS.BB.minRC

Name: La

Class: XmLabel

Templates: StandardFgColor

Current selection

Resource: alignment

```
alignment_beginning
alignment_center
alignment_end
```

Figure 5.32: The Enumeration Box

The Enumeration box has the following fields:

A display box Displays the list of enumerations for the resource named above the box. The selected enumeration is displayed in inverse video.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the selected value by rebuilding, and remains in the Enumeration box.

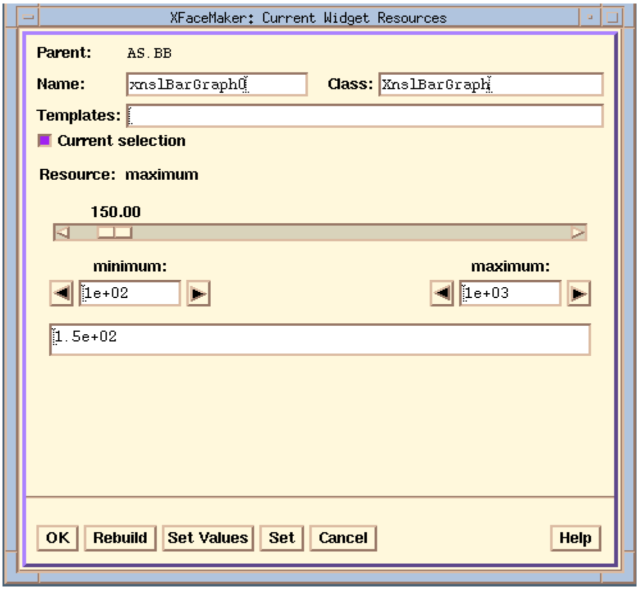
Set Values Applies the change by setting values, and remains in the editor.

Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the Resource window without changing the value.

5.17.6 The Float Box

The Float box is used to set a single-precision floating-point value for a resource. The box is popped up when you click the ellipsis next to a resource that requires a floating-point value.



Parent: AS.BB

Name: xnslBarGraph0 Class: XnslBarGraph

Templates:

Current selection

Resource: maximum

150.00

minimum: 1e+02

maximum: 1e+03

1.5e+02

Figure 5.33: The Float Box

The Float box has the following fields:

A slider Used to specify the value.

A text input box Used to display or specify the value for the resource.

Minimum and maximum arrows Used to set the maximum and minimum values for the resource.

OK Applies the specified value to the resource and returns to the list.

Rebuild Applies the change by rebuilding and remains in the editor.

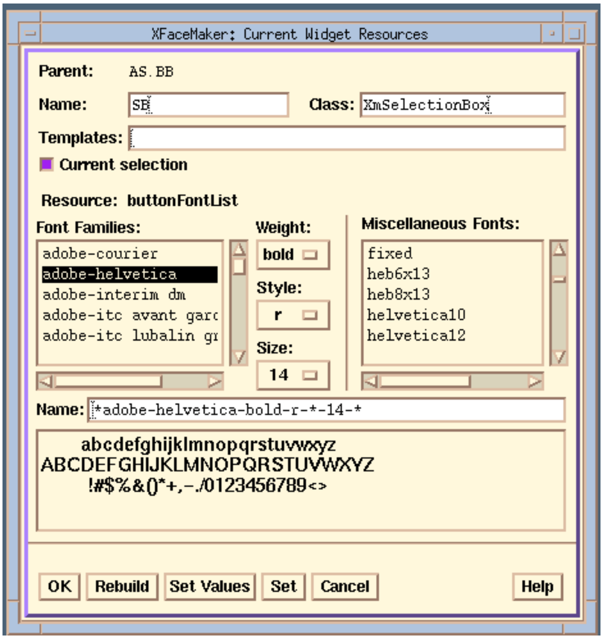
Set Values Applies the change by setting values, and remains in the editor.

Set Updates the value in the Resource list and returns to the list.

Cancel Returns to the list without changing the value.

5.17.7 The Fonts Box

The Fonts box is used to select a font. It is popped up when you select a font resource for a widget, or from inside another editing box; e.g., when you are creating a menu with the Menu Editor box.



XFaceMaker: Current Widget Resources

Parent: AS.BB

Name: SB

Class: XmSelectionBox

Templates:

Current selection

Resource: buttonFontList

Font Families:

```
adobe-courier
adobe-helvetica
adobe-interim dm
adobe-itc avant garc
adobe-itc lubalin gı
```

Weight: bold

Style: r

Size: 14

Miscellaneous Fonts:

```
fixed
heb6x13
heb8x13
helvetica10
helvetica12
```

Name: *adobe-helvetica-bold-r-*-14-*

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
!#$%&()*+,-./0123456789<>
```

Figure 5.34: The Fonts Box

The Fonts box has the following fields:

Font Families Displays a list of the font families currently available for the resource listed above the display field. When you select a font family from this list, a sample of it is displayed in the box below the “Name” field.

Miscellaneous Fonts A list of further available font families. When you select a font family from this list, a sample of it is displayed in the box below the “Name” field.

Weight This option menu lists the weight (or intensity) options for the font characters.

Style This option menu lists the slant options for the font.

Size This option menu lists the type sizes available for the font.

Name Displays the name of the selected font or can be used to enter a font name directly.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Makes the changes by rebuilding, and remains in the editor.

Set Values Makes the changes by setting the values, and remains in the editor.

Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the previous window without changing the value.

5.17.8 The Integer Box

The Integer box is used to edit resources that require an integer value. To open it, click on the ellipsis next to the resource in the Resources list.

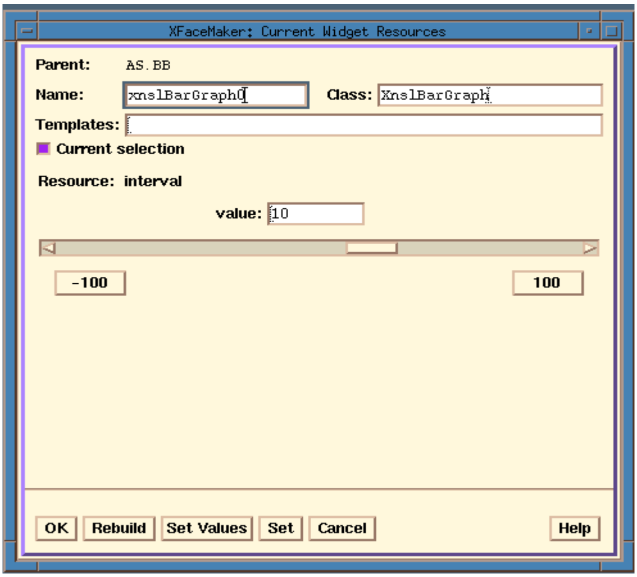


Figure 5.35: The Integer Box

The Integer box has the following fields:

value Used to specify the value for the resource. The slider can also be used to change the value.

Max. and min. buttons Used to set the maximum and minimum values for the resource. Clicking on a button changes the button's maximum or minimum value by a power of 10.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the value by rebuilding, and remains in the editor.

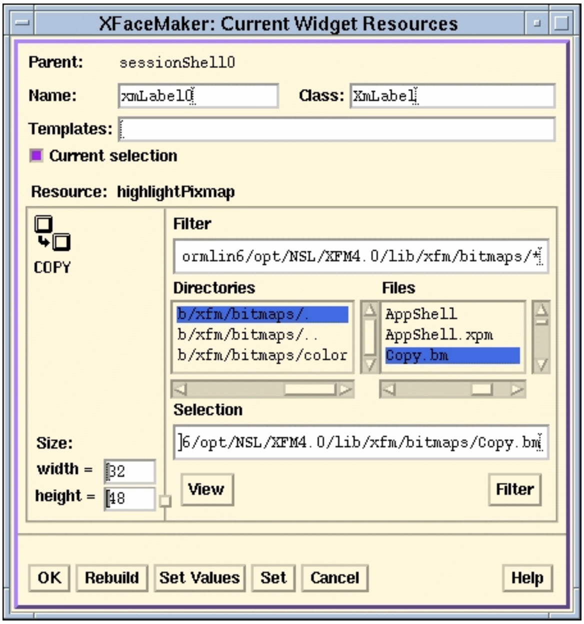
Set Values Applies the change by setting values, and remains in the editor.

Set Updates the value in the list but does not apply it.

Cancel Returns to the main Resource window without value change.

5.17.9 The Pixmaps Box

The Pixmaps box is used to select bitmap or pixmap resources for the selected widget. It is popped up when a pixmap-type resource is selected, and is not available directly from a menu.


XFaceMaker: Current Widget Resources

Parent: sessionShell0

Name: xmLabel0 Class: XmLabel

Templates:

Current selection

Resource: highlightPixmap

COPY

Filter

ormlin6/opt/NSL/XFM4.0/lib/xfm/bitmaps/*

Directories	Files
b/xfm/bitmaps/.	AppShell
b/xfm/bitmaps/..	AppShell.xpm
b/xfm/bitmaps/color	Copy.bm

Selection

]6/opt/NSL/XFM4.0/lib/xfm/bitmaps/Copy.bm

Size:

width = 32

height = 48

Figure 5.36: The Pixmaps Box

The Pixmaps box has the following fields:

A display window The display window displays the current pixmap.

Size The two size fields display the size of the pixmap and are not editable.

File Selection Window This windows enables you to select a pixmap in the standard Motif manner. The window has two buttons:

View To display the pixmap.

Filter To filter the file names. You specify the pattern in the Filter text field above and by pressing the Filter button the corresponding file names are displayed.

OK Registers any changes made to the current pixmap and applies them to the current resource before closing the Images box.

Rebuild Applies the changes by rebuilding, and remains in the editor.

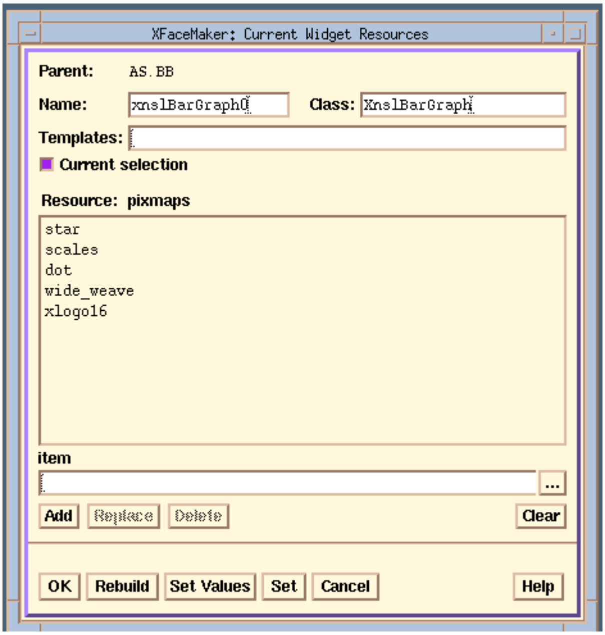
Set Values Applies the changes by setting the values, and remains in the editor.

Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the main Resource window without changing values.

5.17.10 The Pixmaps Table Box

The Pixmaps Table box is used to enter a list of pixmaps for a resource. When a resource needs several pixmaps, you can enter a comma separated list of pixmap names directly in the resource editing field. Or, you can click on the ellipsis and open the Pixmaps Table box.


XFaceMaker: Current Widget Resources

Parent: AS.BB

Name: xnslBarGraph0 Class: XnslBarGraph

Templates:

Current selection

Resource: pixmaps

```
star
scales
dot
wide_weave
xlogo16
```

item

Figure 5.37: The Pixmaps Table Editor

The Pixmap Table box has the following fields:

A List of pixmaps Displays the current list of pixmap names.

A Text Input Field Enter the name of a pixmap here, or click the ellipsis to open the Pixmaps box.

Add Adds the pixmap name of the textField to the list.

Replace Sensitive only if a name is selected in the list. Replaces the name of the selected pixmap with the one in the textField.

Delete Sensitive only if a name is selected in the list. Removes the selected pixmap form the list.

Clear Sensitive only if the list is not empty. Removes all the items from the list. A warning dialog is popped asking for confirmation. Click ***No*** if you do not want to remove all items from the list.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the value by rebuilding, and remains in the editor.

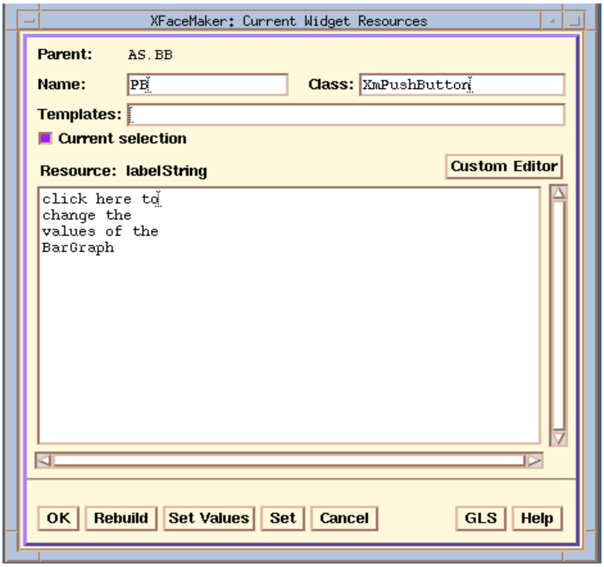
Set Values Applies the change by setting values; remains in the editor.

Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the main Resource window with no value change.

5.17.11 The String Box

The String box is used to enter text for a resource. It is popped up when you select a resource of type String or XmString.



```
click here to
change the
values of the
BarGraph
```

Figure 5.38: The String Editor

The String box has the following fields:

A text input field Used to enter a text string for the widget resource.

Custom Editor Calls the editor (if any) you specified with the FMEDITOR environment variable.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the value by rebuilding, and remains in the editor.

Set Values Applies the change by setting values; remains in the editor.

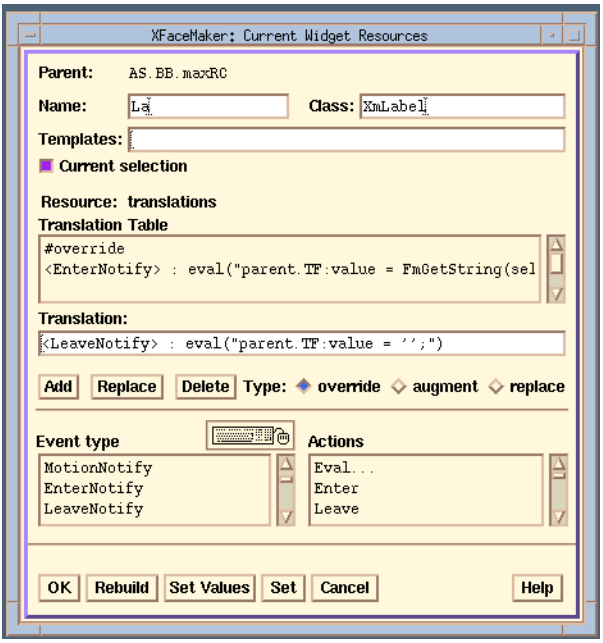
Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the main Resource window with no value change.

GLS Marks the string as an internationalized string, and gives it the default set number. If no message number was specified, the message will be given a number when the messages are saved.

5.17.12 The Translations Box

The Translations box is used to select and create elements for the translation table of the currently selected widget. When you use this box, the required syntax elements are inserted automatically, and XFaceMaker makes sure that they occur in the right order. This box is popped up when you edit a resource of type accelerators or translation.



Parent: AS.BB.maxRC
Name: La Class: XmLabel
Templates:
Current selection
Resource: translations
Translation Table

```
#override
<EnterNotify> : eval("parent.TF:value = FmGetString(sel
```

Translation:

```
<LeaveNotify> : eval("parent.TF:value = '';")
```

Figure 5.39: The Translations Box

Type Specifies how the translations are added to the table.

- **augment** merges new translations into the existing (default) translation table, ignoring translations that conflict with existing ones.
- **override** (the default mode), merges new translations into the existing (default) translation table, new translations overriding any conflicting translations.
- **replace** replaces the existing translations table with the new translations specified

Note: You should check with the local widget documentation that the above modes are valid.

Translation table Displays the translations specified in XFaceMaker for the currently selected widget and the mode: the default mode is override. Clicking on a translation in the list places it in the Translation field. You can then modify the translation and add it to the list by selecting it and clicking on the ***Add*** button, or delete it by selecting it and clicking on ***Delete***.

Translation A text field for entering a translation script.

Add Adds the specified translation to the table.

Replace Replaces the selected translation with the one specified.

Delete Deletes the specified translation from the table.

Event type Displays the standard event types for the currently selected widget. Clicking on an event type displays it in the Translation text line for editing.

Actions This scrolled window displays the standard actions for the currently selected widget. Clicking on an action displays it in the Translation text field for editing. To specify multiple actions, type them in the Translation editing field, separated by commas. The **Eval** action, listed at the top of the list, was added by XFaceMaker. It lets you enter a FACE script as the action. Clicking on **Eval...** opens the Eval Editor.

icon The icon below the Event types window represents a mouse and keyboard. If you do not know the official name of the event, placing the cursor on the mouse-and-keyboard icon, and then simulating the event you are interested in, for instance clicking a mouse button, or pressing a key, will insert the name of this event in the Translation text input line.

OK Applies the specified value to the resource and returns to the main Resource window.

Rebuild Applies the selected value and remains in the editor.

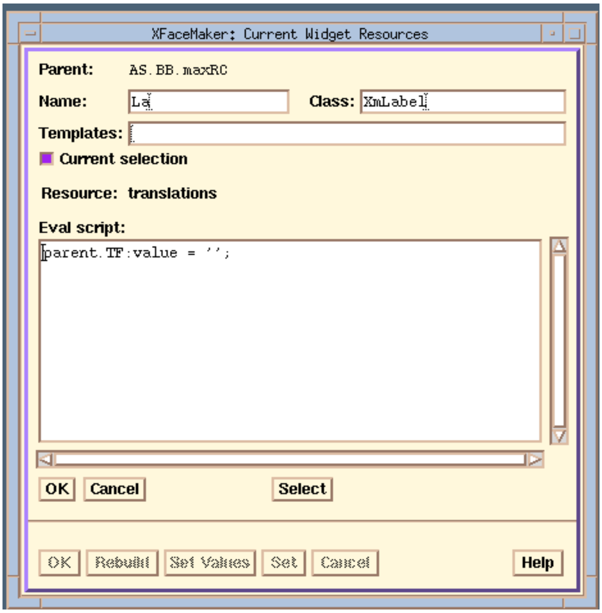
Set Values Applies the change by setting values, and remains in the editor.

Set Updates the value in the Resources list, but does not apply it.

Cancel Returns to the main Resource window with no value change.

5.17.13 The Eval Script Editor

The Eval action is a special case because it invokes a user-defined script instead of a built-in function. In this context, $ is the address of the X event structure. Selecting the ***Eval...*** action pops up a dialog box in which you can enter a standard FACE script. The script can call all the usual FACE functions and any application functions needed. In particular, it can call the function XtCallCallbacks and thus associate a callback to a particular key combination in addition to the standard one.



Parent: AS.BB.maxRC
Name: La
Class: XmLabel
Templates:
Resource: translations

```
parent.TF:value = '';
```

Figure 5.40: Eval Script Editor

The editor has the following fields:

Eval Script A text field for entering a FACE script.

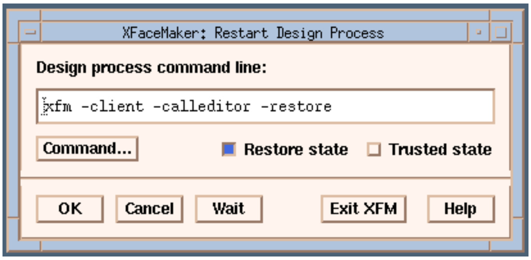
OK This registers the script and applies it to the widget.

Cancel Returns to the Translations window without applying the script.

Select The cursor changes to a “?”. Move it over a widget or widget name and click. This fetches the widget name and enters it in the script editing window.

5.18 The Restart Design Process Box

When the Design process terminates, the Restart Design Process window pops up, displaying the default command line. The default command line launches the Design process which just terminated, usually xfm (or a new user-built instance of XFace-Maker), with all the command-line options specified initially, plus the -client option and the -restore option.


XFaceMaker: Restart Design Process

Design process command line:

```
xfm -client -calleditor -restore
```

Figure 5.41: The Restart Design Process Box

The ***Command...*** button pops the File Selector to choose a different XFaceMaker executable.

The ***Restore state*** and ***Trusted state*** buttons toggle the -restore and -trusted options respectively in the command line.

The ***OK*** button launches the specified command line. XFaceMaker waits until the new Design process connects to it.

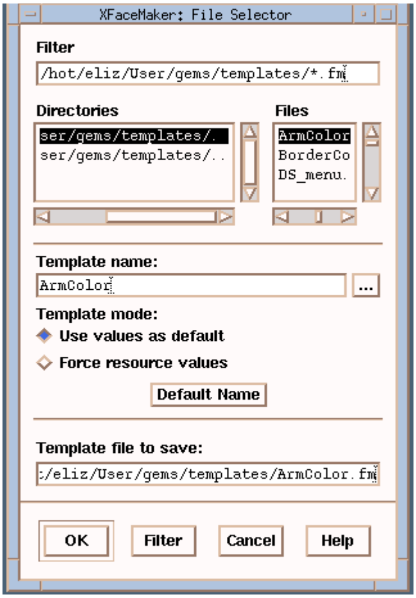
Cancel will not launch any Design process. An alert box pops up to warn you that XFaceMaker is running in mono-process mode. Clicking ***No*** in the alert box returns to the Restart Design Process box.

The ***Wait*** button does not launch any Design process, but XFaceMaker will still wait for a connection. The interface is disabled until the connection occurs. This is useful for launching the Design process manually from a shell, or by any other external means. Don't forget to set the -client option when launching the Design process.

Exit XFM Selecting this button exits XFaceMaker.

5.19 The Template Box

The Template Mode box, is used to specify the mode and/or a name for a template. It is popped up when you edit a template using the commands in the Modules menu, or when you apply a template.


XFaceMaker: File Selector

Filter: /hot/eliz/User/gems/templates/*.fm

Directories:
ser/gems/templates/.
ser/gems/templates/..

Files:
ArmColor
BorderCo
DS_menu.

Template name: ArmColor

Template mode:
[✓ Use values as default]
Force resource values

Template file to save: :/eliz/User/gems/templates/ArmColor.fm

Figure 5.42: The Template Box

The Template Mode box has the following fields:

Use resource values as default — Resources take the default value specified in the template, but can be changed.

Override (force) resource values — Resources are set to the value specified in the template and cannot be changed.

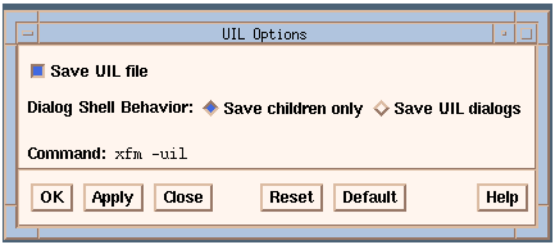
Default Name Clicking here gives the template file name to he template, starting with an uppercase letter.

OK Registers the mode for the selected template and closes the box.

Cancel Closes the Template Mode box without making any changes.

5.20 UIL Options

The UIL Options box is used to set the options for saving a UIL file. All options have an equivalent command line which is displayed in the **Command Line** field.



[✓ Save UIL file]

Dialog Shell Behavior: [✓ Save children only] Save UIL dialogs

Command: xfm -uil

Figure 5.43: UIL Options

The box has the following fields:

Save children only Generates all dialogs as XmDialogShell's with a XmMessage-Box widget child.

Save UIL dialogs Generates predefined combinations of objects called *Dialogs* in the UIL file. These use the Motif convenience functions e.g., XmCreateWarning-Dialog.

Command Displays the command lines for the selected options.

OK Registers the option selected and closes the box.

Apply Applies the selected option and updates the command field without closing the box.

Close Closes the box without changing the options.

Reset Sets the options to the values validated with the last Apply.

Default Returns the options in this box to their default values.

CHAPTER 6: The FACE Language

FACE is a language whose syntax is very close to that of C. It was designed specifically to program the dynamic part of a user interface, and not as a general–purpose language. In contrast to ordinary C, FACE scripts are not compiled while in the development phase of interface construction but are *interpreted*, so that the typical compile-execute cycle is not required. The FACE interpreter reads and executes one statement at a time until the end of the script is reached.

The FACE interpreter is used in several NSL products. This chapter describes only the general features of the FACE language. For information specific to the use of FACE in XFaceMaker, see the next chapter.

6.1 Keywords in FACE

The following identifiers are reserved for use as keywords, and may not be used otherwise:

if else while for

script

self parent toplevel root

global local auto extern

type cprototype

function return

define include

resource

resource is reserved for future use. auto and extern are synonymous with local and global. Avoid using auto and extern in any new scripts that you write. They have been kept for backward compatibility only. The other keywords are described in this chapter.

6.2 Special Characters

The following characters have special meanings in FACE:

; . , : $ @ ‘’ " { } ( ) < > = ! & | + - * /

6.3 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

6.4 Arithmetic Expressions

Arithmetic expressions have the same syntax as in C. The following operators are supported:

Assignment operator	=					
Unary operator	-					
Arithmetic operators	+	-	*	/		
Logical operators	&&	||	!			
Relational operators	==	!=	<=	>=	<	>

All arithmetic operations work on integer or floating-point operands. If any of the operands is a floating-point value, then the operator performs a floating-point operation, but the other operands are *not* converted automatically to a floating-point type (you must use the function citof). Expressions can be parenthesized and operator precedence is as in C.

6.5 Include Files

A file may be included in a FACE script using the include keyword, followed by a file name. i.e.,

```
        include "project.face";
```

The included file may contain any FACE statements. The including script is parsed as if the contents of the file had been included in the script.

6.6 Basic Objects

FACE has the following kinds of objects:

- constants;
- variables;
- special variables;
- widgets;
- widget resources;
- active value variables (when used with XFaceMaker);
- X properties;

6.6.1 Object Types

The type of objects in FACE is specified by a string whose content is arbitrary; i.e., an application may define new variable types if it so desires, as long as they satisfy the type checking and/or type conversion done by FACE when a variable is used. The usual type strings are defined in <Xm/Xm.h>. They correspond to frequently used widget resource types, such as “Int”, “String”, “Pointer” etc. In addition FACE has some special types, described in the next section.

All FACE objects are stored by the FACE interpreter in equally sized memory units which have the size of the X Toolkit-defined XtArgVal typedef, which is usually defined as a C long. When the actual size of the memory needed to represent an object is greater than the size of an XtArgVal (e.g. character strings), a pointer to the object is stored instead.

Most common FACE objects have the same representation as in C: integers are represented as C ints, floating-point values are represented as floats, character strings are represented as pointers to null-terminated character arrays. Only FACE arrays have a special representation.

FACE checks the types of objects as follows:

- when a function is called, the types of the actual parameters are checked against the expected types;
- when a widget's resource is set or retrieved, the type of the value is checked against the type of the resource (except for String values which are automatically converted: see Section 6.6.8);
- when a value is assigned to a statically typed variable, the type of the value is checked against the type of the variable.

FACE accepts a number of resource types as equivalent to the type Int: they are Short, Boolean, Position and Dimension. In addition, when used with XFaceMaker and Motif, all Motif types ending with Position or Dimension (e.g. VerticalDimension) are also equivalent to type Int. Similarly, the type Pixmap and all its Motif variants like XmBackgroundPixmap are equivalent. Finally, FACE accepts the types Window and MenuWidget as equivalent to Widget.

Unlike C, when variables are used or declared, their type need not be specified. If the type of a variable is not specified when the variable is declared, the type is determined each time the variable is assigned and it takes the type of the function or widget resource to which it is assigned. For example:

```
X1 = self:x;
```

self:x is a widget resource. Its type is Dimension. With this assignment, X1 is defined as being of type Dimension.

```
a = f(0);
```

The type of the variable a is the type of the function's return value. FACE checks the type of variables in two instances; when assigning a variable, and when passing it as a function parameter.

6.6.2 Special FACE Types

You can prevent argument type checking by using the special types “None”, and “Any”:

None If, when *attaching* a function, i.e., in FmAttachFunction, you declare the type of an argument as “None” you are in effect telling FACE that the argument type for this argument is not to be checked whenever the function is called.

Any If, when *calling* a function whose arguments have all been specified, you use a type “Any” for one of the arguments, the type of that argument is not checked for that call.

Obviously, these two types should be used only rarely and with care, since circumventing type checking prevents the detection of possible errors.

6.6.3 Constants

Integer constants begin with an optional - (minus) sign followed by a sequence of decimal digits 0-9. FACE integer constants can only be input in decimal form. They have the FACE type Int.

Floating-point constants begin with an optional - (minus) sign followed by a sequence of decimal digits 0-9, a . (dot) character and a second sequence of decimal digits. FACE floating-point constants do not have an exponent part. They have the FACE type Float.

FACE character string constants are placed within optional single or double quotes. A string which does not include blanks does not *have* to be placed between quotes but it is good practice to always quote strings: if a variable with the same name exists, an unquoted string is assumed to refer to the variable though you may have intended it to be a string. Strings which contain blank characters *must* be quoted. The single and double quote characters may also be escaped within a string with \ (backslash). A quoted string may contain the “other” quote character unescaped.

The memory holding a string constant is allocated by the FACE interpreter the first time the script in which the string appears is parsed, and the same memory is re-used each time the script is interpreted. Thus, modifications made to FACE string constants remain in place. FACE character string constants have the FACE type String.

These are examples of FACE integer, floating-point and character string constants:

```
1234
-5678
12.34
-56.78
Hello
"Hello world"
'Hello world'
"Awakening he said: \"Hello world\"."
'Awakening he said: \'Hello world\'.'
"Awakening he said: 'Hello world'."
'Awakening he said: "Hello world".'
```

6.6.4 Variables

Variable names are composed of letters and digits. They must start with a letter and are case sensitive. Thus X and x do not refer to the same variable. All FACE variables have the same size, that of a word or pointer (i.e. the size of an Xt Toolkit XtArgVal), whatever their type. For example, a char variable, which would normally occupy a byte, is converted if necessary to an integer the size of a word.

There are local and global variables. Local variables are known only in the script in which they appear. Thus a local variable may be known in the script of one callback resource but not in another, even of the same widget. You need not declare local variables. They are automatically declared the first time they are used.

```
Message = "Warning: Not found!";
Size = 10;
```

Here, the variable Message has been declared implicitly the first time it has been set. It is of type String. The variable Size is also implicitly declared when first used and is of type Int.

Variables may be declared anywhere in the script, as long as they are not used before they are declared. It is possible to declare a local variable explicitly if it must be used before it is first set.

Local variables are declared using the local keyword:

```
local i;
```

This declares a local variable named i, initializes its value to zero and sets its type to "None". Local variables declared outside FACE functions are *not* automatic variables, i.e., they are not allocated dynamically each time the callback is invoked. They are like static variables in a C function: they are initialized to 0 when the script is parsed and retain their value from one call to another. Local variables declared inside FACE functions *are* automatic, and are allocated and set to 0 each time the function is called.

Global variables can be referenced by all scripts of the same interface. They are declared using the global keyword:

```
    global button_pressed;
```

Global variables can also be declared extern as in:

```
    extern global button_pressed;
```

Extern global variables differ from global variables only in the way they are declared in the C code generated. The `extern' modifier should be used in cases where a global variable is used in several .fm files in order to avoid multiple declarations of the variable in the C code generated: declare the variable as `extern' in all but one script.

Global variables must be declared in all the scripts in which they are used. The parser creates the variable and associated storage the first time it encounters the declaration.

It is possible to declare the type of a FACE local or global variable, either when the variable is declared, with the following syntax:

```
        [local|global] <type> <name>;
```

or when a local variable is first set, using:

```
        <type> <name> = <value>;
```

When a variable is declared with a type, then the variable is *statically typed*, which means that you cannot assign an expression of a different type to it. If you do so, an error message is issued. Static typing is useful for error checking, and also to help the FACE compiler find the type of a variable; e.g., in floating-point expressions.

Note: the keyword auto is equivalent to local, extern to global. Use local and global rather than auto and extern which are kept for backward compatibility only.

6.6.5 Special Variables

Whenever a FACE script is parsed, the two special variables $ and @ are passed to it. @ is the contents of the variable pointed to by $. This is analogous to &x and x (if x is an int), or p and *p (if p is a pointer to an int). The values of $ and @ depend on the context in which the script is being evaluated.

If $ is the address of a structure, then the elements of the structure can be accessed using regular structure addressing constructs, (Section 6.12).[1]

The variables $ and @ have type "Any" which allows them to match any other type wherever they are used. This means, however, that argument and return type checking is not done for $ and @ used in function calls unless they are cast to the proper type; e.g.:

```
    i = atoi(String($));
```

1. FACE also lets you access the structure elements through the @0 to @9 variables where:
@ or @0 is the first element
@1 is the second element
@2 is the third element, etc.
This method should not be used as it is not portable to machines with 64 bit addressing. It has been kept for backward compatibility but any new developements should use standard structure element addressing.

6.6.6 Defined Constants

A constant may be defined with the define keyword. Defined constants are like variables, but their value cannot be changed. Their value is either an integer, a string or a floating-point constant, and is specified after the constant name:

```
    define <name> <value>;
```

Examples:

```
define MAX_WIDGETS 10;
define PI 3.142;
define TITLE "Project X";
self:labelString = TITLE;
```

A defined constant may be defined in several places, but it may not be defined with different values. A defined constant can be used in a script even if it has been declared in another script, provided that the script where the constant has been defined is parsed first. Using the define keyword in the Creation script of the root widget of a hierarchy guarantees that it will be known in all the scripts of the hierarchy.

The FACE interpreter has some predefined constants:

- The constants true, True, false, and False are defined globally. Their type is "Boolean".

  ```
  true  = True  = 1
  false = False = 0
  ```

- When using FACE in XFaceMaker, XFaceMaker also adds most constants defined by the OSF/Motif toolkit.

6.6.7 Widgets

The FACE naming scheme uses the . (dot) operator, which lets you travel from a widget to one of its children. Widget names have the following form:

```
[<reference>.][<name1>[.<name2>]...]
```

where reference is either one of the keywords below, or a variable name, and name1, name2, ... are widget names that form the path of the designated widget in the widget hierarchy.

FACE recognizes the following keywords to designate the reference of a widget name:

self refers to the widget from which the script is called. In the case of FaceEvalString, it refers to the widget that was passed as an argument.

parent refers to the parent of self. In the case of FaceEvalString, it refers to the parent of the widget that was passed as argument to FaceEvalString. Multiple parent. keywords can be concatenated to designate the parent's parent, etc.

toplevel refers to the child shell of the ApplicationShell (root) which is an ancestor of self, or root itself if root has no shell child that is an ancestor of self.

root refers to the ApplicationShell which is an ancestor of self. This is necessary, for instance, to travel from a TopLevel shell to another TransientShell at the same level, e.g. Show(root.TransientShell).

If a variable name is used as the first component of a widget name instead of one of the keywords below, then the variable must contain a valid widget pointer which will be used as the reference widget. The remainder of the widget name, i.e. the path of sub-widget names, is used to find the sub-widget by calling an equivalent of the X Toolkit function XtNameToWidget with the reference widget and the sub-widget path. If there is no sub-widget path, the reference widget is returned.

When naming a widget, the name must be made explicit by starting it with one of the above reserved names.

The following are legal widget names in FACE:

```
self
parent
root
toplevel
self.Wix2
parent.ScrollBar
parent.parent.WarningBox
```

```
root.GrabShell.GrabBoard.Input
toplevel.GrabBoard.Input
parent.GrabBoard.Input
root*Input
wid.Button /* wid is a variable of type Widget*/
```

You can refer to a widget in a FACE script using the wildcard character * in a manner similar to the X Toolkit function XtNameToWidget. The following limitations apply:

- The * can only be used in the sub-widget path part of a widget name, and it cannot be used at the beginning or at the end of a widget name.
- If used in association with a variable name, the variable must be explicitly declared of type Widget. Otherwise, the * will be interpreted as *multiply.*

These are legal examples of the use of the wildcard character in widget names:

```
self*Button
root.Shell*ScrollBar
wid*Text /* wid is a statically typed Widget variable */
```

When naming widgets, do not include a space, a dot, a star, a colon, a semi-colon within the name. Avoid using any reserved C words for widget names: a widget name must be a valid C identifier. XFaceMaker checks widget names as you specify them and will output a message if it thinks a name is not valid.

6.6.8 Widget Resources

Widget resources can be accessed by a widget name, followed by the : (colon) operator, followed by the widget-defined resource name as specified in the widget reference manual. You must use the resource name without its prefix, as in a .Xdefaults resource file; e.g.

“self : x", *not* “self : XmNx".

The following are examples of legal resource names:

```
self:x
self:arrowDirection
parent.parent.button:borderWidth
```

A variable to widget resource assignment is of the form:

```
widget:resource = abc;
```

Each time an assignment statement such as

```
i = self:x;
```

is encountered, the function XtGetValues is called to retrieve the current value of the resource referred to.

Similarly, each time an assignment statement such as

```
self:x = i;
```

is encountered, the function XtSetValues is called to update the current value of the resource referred to.

Note: If you refer to the documentation on widget resources, you will see that the size of variables representing these resources varies according to the resource type. This is not so in FACE where all variables are the same size (that of XtArgVal). This means that resources whose size are smaller than a word are converted to a word-sized integer, and resources of type String are replaced by a pointer to the beginning of the string.

If the right-hand side of a resource assignment is of type “String” and the resource is of another type, an implicit type conversion is done: the FACE interpreter calls the converter registered with the Xt Toolkit to convert the string to the appropriate resource type.

If the right-hand side of a resource assignment is of another type, then this type must be compatible with the type of the resource, or an error will occur. For example:

```
self:width = "1";
```

converts the string "1" to the value 1 before doing the assignment.

```
self:width = "hello";
```

produces an error because the **String** to **Int** conversion function is unable to convert hello to a type **Int**.

```
self:width = 2;
```

does not produce an error and does no type conversion: the type of the variable and the type of the resource are both **Int**.

```
self:labelString = 1;
```

produces an error because the resource type for labelString is **XmString**, not **Int**. In such a case, no implicit conversion function would be called since 1 is not of type **String**.

```
self.arrowButton1:arrowDirection = "arrow_up";
```

converts the string to type **Arrow Direction**.

Note: If you use the predefined OSF/Motif constants as right-hand side values in resource assignments, you might have to cast them to the appropriate type because they are all declared with the type “Int”.

```
type ArrowDirection;
self:arrowDirection=ArrowDirection(XmARROW_RIGHT);
```

FACE accepts a number of resource types as equivalent to an Int: they are “Short", “Boolean", and all Motif types ending with “Position" or “Dimension", such as “VerticalPosition". Similarly, the types “Pixmap" and all its Motif variants such as “BackgroundPixmap", are equivalent.

In a few cases, you may have to specify the type of a resource explicitly using the following syntax:

```
<widget-reference>:<resource-name>:<resource-type>
```

For example, self:value:Int means that the resource value is of type Int. This is useful only to inform the FACE compiler of the type of the resource when generating standalone C code if the compiler cannot determine it by itself.

6.6.9 X Properties

FACE lets you refer to X properties to store data. X properties are variables associated to an X window. For that reason, they can be used only for widgets that have already been realized. Since such a variable is associated to the particular instance of the widget for which you define it, its value is independent of any other. Thus, the value of a property with the same name defined for a different instance may be different.

It is legal in FACE to use properties for gadgets but, since gadgets do not have an X window, the property of a gadget is associated to the gadget's parent. Thus, for gadgets, you must be careful not to use the same property for two gadgets with the same parent, or for a gadget and its parent, if you want the variables to be independent.

To define a property, an assignment statement is required, such as:

```
widget:<property_name> = self:x; where:
```

widget is the name of the *realized* widget for which you are defining a property.

property_name is the name of the property, i.e., the name you will use to refer to that variable. Note that the name is placed in <angle brackets>.

With such a statement, you have defined a new variable name, and its type: the type of the property is the type of the resource or of the variable which is on the other side of the assignment statement.

For example, you can write:

```
self:<area> = self:height * self:width;
parent:<area> = parent:height * parent:width ;
```

to define two variables (properties) called area, both of type **Int**, both being the area of the associated widget. If self is a gadget and not a widget, however, area is a single variable, whose value is the area of parent after the second statement is executed.

6.6.10 Widget Active Values

In XfaceMaker, an object can have active values attached to it. Such active values can be referenced in FACE using the following constants:

```
<widget>:@<avname>   and   <widget>:$<avname>
```

These expressions can be used in the same manner as resource references, but will read or write an active value associated with a given widget.

v = <widget>:@<avname> fetches the address attached to the specified active value and associated with the specified widget (possibly allocating a memory location of size sizeof(XtArgVal) if the active value is automatic), calls the *get* script of the active value with the address passed as the $ argument, and returns the contents of the address. The type of the result is the type of the active value. If the active value does not exist, or if it is immediate, or if it is not attached and it is not automatic, a FACE error occurs.

v = <widget>:$<avname> fetches the address attached to the specified active value and associated with the specified widget, calls the *get* script of the active value with the address passed as the $ argument, and returns the address. The type of the result is the type of the active value if it is immediate, or Pointer otherwise. If the active value does not exist, a FACE error occurs.

<widget>:@<avname> = v fetches the address attached to the specified active value and associated with the specified widget (possibly allocating a memory location of size sizeof(XtArgVal) if the active value is automatic), converts v to the type of the active value if v is of type String, stores v as the contents of the active value address, and calls the *set* script of the active value with the address passed as the $ argument. If the active value does not exist, or if it is immediate, or if it is not attached and it is not automatic, or if v is a String which cannot be converted to the type of the active value, or if v is not a String and the type of v is not compatible with the type of the active value, a FACE error occurs.

<widget>:$<avname> = v changes the address attached to the specified active value and associated with the specified widget, and calls the *set* script of the active value with the new address passed as the $ argument. If the active value does not exist, or if it is immediate and the type of v is not compatible with the type of the active value, or if it is not immediate and the type of v is not compatible with the type Pointer, a FACE error occurs.

6.7 Type Casters

Type casting functions are available in FACE. They do not *convert* an object, i.e., they do not change its value but they *cast* the object type to another type to satisfy FACE type checking requirements. These functions return their argument, which may be of any type, in the appropriate type. FACE type checking does not consider all resource types as distinct. For example, types “Position”, “Dimension”, “Boolean” are all equivalent in FACE.

6.7.1 Declaring New Type Casters

New type casting functions can be declared using the keyword type followed by the variable type to which you want to be able to cast. After such a declaration, the new type casting function exists. For example:

```
type NewType;
n = NewType(x);
```

6.7.2 Declaring Enumerated Resource Types

The type keyword is also used to declare a new enumerated type along with its values with the following syntax:

```
type <type-name> = <value0>, <value1>, ... ;
```

This declares a new type caster, and defines constants for every value.

For example:

```
type Fruit = apple, orange, banana;
```

declares the new type caster Fruit, and defines 3 constants:

```
define XfmAPPLE 0
define XfmORANGE 1
define XfmBANANA 2
```

In addition, a new type converter is registered in the X Toolkit so that the strings "apple", "orange" and "banana" can be converted to the corresponding constant.

The prefix of the new array can be specified like this:

```
type Fruit = [Xyz] apple, orange, banana;
```

This would define the contents:

```
define XyzAPPLE 0;
```

etc.

6.7.3 Explicit Type Conversion

An explicit type conversion function is available to convert a string to the type specified as one of the arguments:

ConvertString(Widget *w*, String *S1*, String t*arget_type*);

where:

w is the widget used for additional arguments, if required, as in the XtConvert function.

S1 is the string you want to convert.

target_type is the string specifying the type you want to convert to.

The return type is type “Any”. For example:

```
ConvertString(w,"10","Int");
```

returns the value 10 of type “Any”.

6.8 Statements

A FACE script is a sequence of one or more statements. Each statement is either:

- an assignment
- a control structure
- a function call
- a call to a named script.

6.8.1 Assignments

Assignments are, as in C, of the form:

```
<left-hand-side> = <right-hand-side> ;
```

The left-hand side of an assignment can be:

- a variable (local or global),
- a widget resource,
- a widget property,
- the special variable @,
- an active value of the form @name, widget:@name, or widget:$name (in XFace-Maker).

The right-hand side of an assignment can be any expression, that is:

- a constant (integer, character string or floating-point),
- a variable,
- a widget reference,
- a widget's resource,
- a widget's property,
- a function call,
- a special variable ($ or @),
- an active value's address $name (or widget:$name), or contents @name (or widget:@name) (in XFaceMaker),
- an arithmetic expression, which itself can be composed of arbitrary expression operands.

6.8.2 Control Structures

FACE provides the if–else, while, and for statements. These have the same syntax and semantics as in C. The following example illustrates the use of if–else:

```
valueChangedCallback =
        Loudness = self:value;
        if (Loudness == 0)
                ResetVolume();
        else if (Loudness < 10) {
                Slide = Slide + Slide/10;
                SetVolume(Slide);
        } else
                SetVolume(Slide);
```

FACE provides a while loop equivalent to that used in C. For example, the following code performs a linear search in an application database for information keyed on a string selected from a TextList:

```
                   Running = 1;
        ResetTable();
        while (Running) {
                Key = GetTable();
                Running = equal(Key, $);
        }
        DisplayCurrentTable();
```

For example:

```
Int a[String];
...
for(String s = first(a, 0); s != 0; s = next(a, s, 0)){
printf("element %s = %d\n", s, a[s]);
}
```

6.8.3 Function calls

Function calls have the same syntax as in C:

```
<function_name> ( <arg1> , <arg2> ... ) ;
```

A function call can be used as a statement by itself, in an expression, or as the right-hand side of an assignment. When a function call is used by itself, its return value is lost.

FACE checks the types of the arguments when calling a function, as explained in Section 6.6.1. An argument type mismatch causes an error, and execution aborts.

The current implementation of the FACE interpreter uses the "traditional" function calling conventions. A function call is performed by the FACE interpreter through a function pointer declared in C as follows:

```
typedef int(*FaceFunctionPtr)();
```

This means that the function pointer is *not* prototyped as it could be with an ANSI C compiler, so all arguments are passed to the function using the default argument promotion rules used by non-ANSI C compiler:

- chars, shorts, and ints are passed as longs,
- floats are passed as doubles.

For this reason, the FACE interpreter only distinguishes between float and integer (or pointer) arguments when calling functions (see Section 6.10). In addition, pointers are passed as longs.

6.8.4 Named Scripts

The keyword script must always be followed by an identifying *name*, and has two uses:

- to name scripts,
- to share scripts.

The *name* must begin with an alphabetic character.

The keyword script labels and executes a named script if one is defined immediately after a : (colon), or executes the named script if no colon announcing a script definition follows (i.e., it is followed by an end of statement ; (semicolon)). The first time FACE encounters a script, it caches the script, using its name as an index. Therefore two different scripts should never have the same name or the second one will not be executed.

Eval scripts, should be named systematically even if they are only used in one place, to make use of this caching mechanism. A *named* Eval script is parsed only the first time it is encountered whereas *un–named* Eval scripts are parsed each time they are executed, thus increasing memory allocation.

The second usage is to share a script. For example, if two buttons in an interface are to have the same help facility, the first defines the script named help:

```
helpCallback =
    script help:
    root.HelpBox:dialogMessage = "help!";
    ret = PopupAndWait(root.HelpBox);
```

while the second simply calls it causing it to be retrieved from the cache:

```
helpCallback =
    script help;
```

The effect will be the same when either callback is called; the values of self and parent are those of the widget which called the callback.

Note however, that a script definition or call can only appear on its own within a FACE script. The following is *not* legal:

```
helpCallback =
    self:insensitive = true;          /* wrong */
    script help;
```

6.9 FACE Functions

It is possible to define C-like functions in FACE, with the following syntax:

```
function Int MyFunction(String s, Boolean b)
{
   if ( b == True )
      return(strlen(s));
   else
      return(0);
}
```

FACE function definitions can be nested, that is, a function can be defined inside another function, but the scope of functions is always the whole interface (functions are declared globally). Recursive calls are also allowed.

FACE functions can be defined anywhere in a FACE script. They can be declared (to be defined later) using the same syntax to declare application functions, as described in Section 6.9.2. When a function is defined or declared several times in FACE scripts, the following rules are applied:

1. An application function (i.e. a function name attached to a real function by the application, or a predefined function) is never overridden;
2. If a function is declared several times with different prototypes, a warning is printed, but the function declaration will be changed, unless it is an application function (first rule).

The return keyword has the same semantics as in C. It must be called with parentheses, and it may be called with zero or one argument (of any type). It can be used in FACE functions, but also anywhere else in a FACE script, in which case it will immediately leave the FACE script (the return value is then ignored). No type checking is done between the type of a FACE function and the argument of return.

6.9.1 Scope of Variables in FACE Functions

In FACE functions, local variables are automatic: a new instance of the variable is created every time the function is called. The scope of variables is defined as follows:

- the scope of a local variable declared outside a function is the whole script, *not including* functions defined in the script;
- the scope of a global variable declared outside a function is the whole script, *including* functions defined in the script (i.e., it is not necessary to declare the global variable again inside the function);
- the scope of a local variable declared inside a function is the function itself, not including other functions defined inside the function;
- the scope of a global variable declared inside a function is the function itself, and the rest of its surrounding script.

When a FACE function is called, the reference widget and call_data (accessed with the special variable $), are those of the script from which the function was called. For example, in:

```
widget XmPushButton {
   activateCallback = function Widget MySelf()
           {
           return(self);
           }
} PB1
widget XmPushButton {
   activateCallback = function Widget MySelf();
           w = MySelf();
} PB2
```

the call to MySelf in widget PB2 returns PB2, not PB1.

6.9.2 Declaration of Application Functions

Application functions are not known to FACE until the application is linked with the interface; as a consequence, any reference to an application function in a script generates an error message. To avoid this, you can declare your application functions, using the keyword function.

The syntax is:

```
function ret_typ fun_name(typ_arg1,typ_arg2,...)[= val];
```

where:

ret_type is the type of the value returned by the function

fun_name is the name of the function

typ_arg1 is the type of the first argument

type_arg2 is the type of the second argument

... further argument types up to eleven.

= val is an optional argument giving the value to be returned during simulations.

Example:

```
function Int my_fun(Widget,Int,String) =1;
```

declares the function my_fun which returns an Int whose value during simulations is 1.

In principle, such a function need be declared only once, the first time it is referred to. When in doubt as to the order in which scripts are analyzed, you can insert the declaration before each reference to the function.

FACE functions can also be declared as global:

```
global function None Hello(String message)
{
    printf(“Hello, %s\n”, message);
}
```

Global FACE functions differ from non-global functions only in the way their are output in the generated C code.

6.10 Floating-Point Variables

Single-precision floating-point values are implemented in FACE. A floating-point constant is of the form i.f, where i and f are decimal numbers. Floating-point values work as in C, with the following restrictions in FACE:

- Only the first six arguments of a function call can be floating-point values;
- There is no automatic conversion between floating-point and integer (or other) values; values can be explicitly converted using the FACE functions cftoi and citof, which convert a float to an integer and vice-versa; the definition of the conversion (truncating or rounding) is the same as the definition of the conversion operator in C, and is machine-dependent;
- An arithmetic operator is assumed to be a floating-point operator if one of its arguments is a float, but the other argument is not converted automatically to float: the converter citof must be used if needed; if a non-float argument is passed to a float operator, a parse error occurs;
- Floating-point arguments are passed correctly to functions only in the following cases:
  - The function is declared as taking an argument of type XtRFloat with FmAttachFunction,
  - The argument is a floating-point literal,
  - The argument is a statically typed variable (or a *defined* constant) of type Float,
  - The argument is a call to a function returning a float.

The last case makes it possible to force XFaceMaker to pass an argument as a float, even if the function was not declared as taking a float, by using the Float type caster.

For example, printf accepts arguments of unspecified type: to pass a float, you must cast it to Float, as follows:

```
pi = 3.1415926;
          printf("pi=%f\n", Float(pi));
```

Another solution is to use a statically typed variable, or a constant:

```
define pi 3.1415926;
     Float pi2 = pi/2.0;
       printf("pi=%f, pi2=%f\n", pi, pi2);
```

If a FACE–attached application function returns a float, it must be declared as returning a value of type XtRFloat in FmAttachFunction, otherwise the FACE interpreter will get an incorrect return value. Similarly, in compiled interfaces, (C-code versions), if an application function returns a float this must be specified in the FACE script using the function keyword:

```
function   Float  myfunction(type1,type2,...)
```

6.11 Using Arrays in FACE

FACE arrays are a form of association table which can be indexed by strings, integers or arbitrary values like pointers. FACE provides two kinds of arrays: hash arrays and linear arrays.

A FACE hash array is a hash table which can be indexed by a string key or by an arbitrary one-word key. In a string-indexed hash array, an element is stored and accessed using the value of a null-terminated character string: two strings which are lexically equal will give access to the same element, even if they are stored at different addresses. In a value-indexed hash array, only the immediate value of the index is used, and only two values which are arithmetically equal give access to the same element.

A FACE linear array is similar to a C array. The elements are stored in a contiguous area of memory, and are indexed by integers only. The memory for a linear array is allocated and expanded automatically as needed. FACE linear arrays can give access to C arrays defined by the application or by Motif widgets, for example the array of items in an *XmList*.

6.11.1 Declaring Arrays

FACE arrays are special variables which must be declared with the following syntax:

```
[<element-type>]
<variable-name> "["[<key-type> [","<default-value>]]"]"
```

element-type is the FACE type of the elements of the array; if an element type is specified, the array elements will behave like statically-typed variables with type checking, otherwise, any type can be stored in the array, and all array elements will have the type Any;

variable-name is the name of the FACE variable which will hold the array;

key-type is the FACE type of the keys used to index the array; if the key type is String, then the array will be indexed by strings, otherwise it will be indexed by values. If the key type is not specified, it defaults to None, and the array is indexed by values; for linear arrays, the key type must be Int;

default-value allows to specify a value which will be returned when an uninitialized element is accessed; this value must be a constant; it is 0 if not specified.

For example, here is how to declare a hash array a of integers indexed by strings and with a default value of -1:

```
Int a[String,-1];
```

An array can be declared anywhere, as for other FACE variables, and can be declared local or global. The scope rules are the same as for ordinary variables.

FACE function parameters can be declared as arrays, for example:

```
function String Foo(String a[Int], Int i)
{
    return(a[i]);
}
```

Array variables can be used as any other variables. In particular, they can be passed as parameters to functions:

```
String a[Int];
...
Foo(a, 10);
```

The same array can be accessed using different array variables declared with different types. For example, say you want to access a hash array sometimes by strings, and sometimes by values. To do this, you can declare two array variables and assign the value of one to the other:

```
Int indexed-by-ints[Int];
...
Int indexed-by-strings[String];
...
indexed-by-strings = indexed-by-ints;
...
```

It is also possible to declare an array variable without actually declaring the array. To do this, declare an ordinary variable of type FaceArray:

```
FaceArray a;
```

Such a variable can hold the value of any array, but it cannot be used to access the elements of the array.

6.11.2 Allocating Arrays

Declaring a FACE array does not allocate any storage for it: the variable is initialized to 0. Before an array can be used, it must be allocated by calling the new_table, new_array or new_c_array functions.

The new_table function creates a new hash array. It takes no arguments.

```
Int a[String];
a = new_table();
```

The new_array function creates a new linear array. The storage for the elements will be allocated and expanded automatically when required:

```
Int a[Int]
a = new_array();
```

The new_c_array function also creates a new linear array, but allows you to initialize the array with a C array. It takes two arguments: the first argument is of type Pointer and specifies the C array; the second argument is of type Int and specifies the size of the C array.

```
XmString items[Int]
items = new_c_array
  (Pointer(parent.list:items), parent.list:itemCount);
```

The C array passed to new_c_array must be an array of elements of the size of an XtArgVal. An initialized array is *not* expanded automatically. When the array is not used any more, it should be freed by calling the destroy function:

```
destroy(a);
```

When an initialized linear array is freed, the C array passed to new_c_array is not freed. FACE arrays are allocated from the heap, so the storage of an array remains valid, regardless of the scope of the variable which holds it, until a call to destroy is made. In particular, an array allocated in a function is not freed automatically when the function exits.

6.11.3 Accessing Array Elements

Array elements are accessed with the following syntax:

```
<array-variable> "[" <key> "]"
```

Array elements can be used in expressions and as the left-hand side of assignments:

```
Int a[Int] = new_array();
a[10] = 20;
printf("element no. 10 of array a is: %d\n", a[10]);
```

Array keys can be arbitrary expressions. If the array has been declared with a key type, the type of the key is checked against the declared key type. In an assignment, if the array has been declared with an element type, the type of the right-hand side is checked.

Writing to a non-existing entry of a hash array creates a new entry. Writing beyond the current bound of a linear array expands the array automatically, except if the array was initialized using new_c_array. If the index used to write to a linear array is greater than the last index (i.e., greater than or equal to the size of the array), the array is also expanded automatically, but any "gap" is filled with zeros, (*not* with the array's default value).

Reading from an array element which was not previously written to, returns the default value specified when the array was declared, or 0 if no default value was specified. Reading from an unallocated array (i.e. an array variable with a null value) also returns the default value. Writing to an unallocated array does nothing.

6.11.4 Iterating Through Arrays

There are two functions which allow the programmer to iterate through the elements of an array.

- The first function returns an initial key for an array. It takes two parameters: an array variable, and a key value which will be returned if the array contains no elements.
- The next function returns the *next* key in an array. It takes three parameters: an array variable, the previous key returned by first or next, and a key value which is returned if the end of the array is reached.

  ```
  Int a[String];
  ...
  for(String s = first(a, 0); s != 0; s = next(a, s, 0)){
  printf("element %s = %d\n", s, a[s]);
  }
  ```

Notes:

- It is guaranteed that the key for each element of the array will be returned exactly once by the first and next functions, but for a hash array the order in which the keys are returned is **not** specified. For a linear array, though, the indexes are returned sequentially.
- The value passed to first and next as the end value must not be the key for a valid element of the array.

These iteration functions are most useful with hash arrays. To iterate through linear arrays, one can simply increment an integer index and use the size function.

6.11.5 Getting the Size of an Array

The size function returns the number of elements of an array. For hash arrays, it is the number of values stored in the array. For linear arrays, it is the largest index used to store a value minus one, or the number of elements specified for an initialized array.

6.11.6 Getting the Address of an Array

The function data can be used to pass the address of a linear FACE array, (that is, an array created with new_array or new_c_array), to an application function. data returns a pointer to the actual data contained in the array. It takes one argument, which is the variable holding the array. For example:

```
String t[Int] = new_array();
t[0] = "Hello";
t[1] = "World";
...
MyFunction(data(t));
```

MyFunction receives a regular C array.

6.11.7 Clearing an Array

The clear function deletes all entries in an array, but not the array itself:

```
clear(a);
```

The array is then as if it had just been allocated. When an initialized linear array is cleared, the C array passed to new_c_array is not freed. It is not possible to delete a particular entry of an array; instead, the default value of the array can be stored as the element value. In that case, the first or next functions will return a valid key which will yield the default value.

6.12 Structures

A FACE structure is similar to a C structure, except that FACE provides access only to structure *pointers*. A structure can be declared either from the C code of an application, or inside a FACE script.

6.12.1 Declaring a Structure from a C application

To declare a structure of your application which you want to be able to use in FACE scripts, use the FmRegisterStructures function.

```
int FmRegisterStructures(
        FmStructDesc *structs,
        int num_structs)
```

This function declares the structures described by *structs*, which points to the following type:

```
typedef struct _FmStructDesc {
        String name;
        String cname;
        FmFieldDesc *fields;
        int num_fields;
        String id_field;
        XtArgVal id_value;
} FmStructDesc;
```

name is the name of the structure, i.e. the FACE type which will be used to declare pointers to the structure. This can be different from the actual C identifier.

cname is the C name which will be used to generate references to the structure in the C code generated by XFaceMaker. It is the declaration that would be used in a C file to declare a pointer to the structure.

fields is an array of FmFieldDesc structures:

```
typedef struct _FmFieldDesc {
        String name;
        String type;
        int size;
         int offset;
}FmFieldDesc;
```

where:

name is the name of the field (this must be the same as the C identifier for the structure field);

type is the FACE type of the field;

size is the size in bytes of the field;

offset is the offset in bytes of the field from the beginning of the structure.

num_fields is the number of elements in fields.

id_field is currently unused, and must be set to 0.

id_value is currently unused, and must be set to 0.

The offset of a structure field can be computed using the XtOffset macro contained in X11/Intrinsic.h. The size can be computed using the FmSize macro, which accepts the same arguments as XtOffset, and which is defined in Fm.h.

6.12.2 Declaring a Structure in a FACE script

You can also declare a structure in a FACE script with this syntax:

```
struct <struct-name> [<c-struct-name>] {
<field-type> <field-name>;
...
};
```

For structures declared in FACE scripts, all fields have a size of one FACE word. The C pointer type c-struct-name is optional. If it is specified, it must be defined in a header file included in the C file generated by XFaceMaker using the -cinclude option.

6.12.3 Declaring Structure Variables

Variables which are to hold a structure pointer must be declared with the type of the structure, and can be used as ordinary statically typed variables. For example, in the activateCallback of a push button, write:

```
XmPushButtonCallback cbs;
cbs = $;
```

6.12.4 Referencing Structure Fields

The fields of a structure are accessed using the -> operator. The resulting value has the type of the field, as declared in the structure descriptor.

```
XmPushButtonCallback cbs;
cbs = $;
printf("clicks: %d\n", cbs->click_count);
cbs->click_count = 0;
```

In general, to use callback structures, use XmxxxCallback and *not* XmxxxCCall-backStructPtr.

6.12.5 Allocating Structures

A new structure can be allocated in a FACE script using the function new_struct, which takes the name of the structure as parameter:

```
struct MyStruct {
Int i;
String s;
};
MyStruct ms = new_struct(MyStruct);
ms->i = 1;
ms->s = "Hello";
```

Structures are allocated from the heap, so there is no automatic de-allocation when a function exits. A structure allocated using new_struct can be freed when no longer needed using free or XtFree.

6.12.6 Arrays as Structure Elements

This example shows how to declare a structure which contains a FACE array, and initialize the array:

```
struct Mystruct {
Int i;
FaceArray a;
Int j;
};
global Mystruct s;
s = new_struct(Mystruct);
s->a = new_array();
String a[Int] = s->a;
a[0] = "zero";
a[1] = "one";
a[2] = "two";
a[3] = "three";
```

CHAPTER 7: FACE in XFaceMaker

In XFaceMaker, FACE scripts can be used in:

callbacks: any of the callback resources associated to widgets. A script associated to a callback is launched when the callback is activated. See the documentation for each widget class.

creation scripts: a special resource associated by XFaceMaker to all widgets, called xfmCreateCallback. A script associated to a widget's xfmCreateCallback is launched at the time of widget creation, after the widget and all its children have been created.

active value Set and Get scripts: active values can be used to share data between the application and the interface. Active values are named variables associated to a widget. It is possible to associate two scripts to each active value: a Get script, usually used to fetch a value from the interface and a Set script, usually used to set a value in the interface. Active value Set and Get scripts are triggered from the application or the interface, via the special functions SetActiveValue and GetActiveValue.

translation eval scripts: XFaceMaker adds a special action, called Eval, in which a FACE script can be entered.

the application: by calling the application function FaceEvalString, the application can directly specify and launch a FACE script.

startup script: the startup file .xfm.startup is a FACE script read by XFaceMaker to determine the user's preferences for the position and state of various windows. You can add extra commands to this file as desired. However, they will be lost if the *Save Prefs* command is used.

widget class creation: FACE scripts are used to specify side effects in widget class methods, as well as in the definition of new resources when you create a new widget class using XFaceMaker.

7.1 Value of $ and @ in XFaceMaker Scripts

The value of the FACE special variables, $ and @ which are passed to a script when it is launched depends on the context of the script as follows:

callbacks In a callback script, $ is a read-only variable referring to the call_data of the widget whose callback is being executed. With the OSF/Motif toolkit, this is the callback structure. The elements of the callback structure can be accessed with regular structure addressing. For example, in the valueChangedCallback of a ScrollBar, the callback structure has the following elements:

```
int         reason;
Xevent      *event;
int         value;
int         pixel;
```

To print the scrollbar value, you could have the following script:

```
XmScrollBarCallbacks;
cbs = $;
printf("new value = %d\n", cbs->value);
```

creation scripts: In a creation script, the value of $ is 0 and the use of @ is forbidden.

active value Set and Get scripts: In active value Set and Get scripts, @ is the value and $ is the address of the active value in the widget whose active value script is being parsed (the *current* widget).

translation eval scripts: In a translation eval script, $ is a pointer to the Xevent which triggered the execution of the script. The structure elements can be accessed using regular structure addressing.

For example, in a translation eval script associated to a ButtonPress event, to print which button was pressed, you could have the following script:

```
XButtonEvent xev;
xev = $;
printf('button %d pressed\n',xev->button);
```

the application In scripts launched using the function FaceEvalString, $ is the address of the param argument; @ is the content of that address.

startup script In startup scripts, the use of $ and @ is forbidden. These two special variables are reserved for internal use.

widget class creation In scripts associated to widget class methods, $ is a pointer to a structure in which the parameters passed by the X Toolkit are passed on to the script. There is a different structure for each method. The Class Method Arguments table summarizes the parameters passed. Refer to the section on widget class methods in Chapter 11 for further details.

7.2 Active Value Variables

XFaceMaker, through the active value mechanism, lets the application know about variables used in the interface. An active value is the association of a named variable, a widget, a Set script, a Get script and, once the active value is attached to the application, an application address.

Active value names are composed of letters and digits. They must start with a letter and are case sensitive. XFaceMaker lets you refer to their value and address directly as follows:

@Name refers to the content of active value “Name"

$Name refers to the address of active value “Name"

The Set script of an active value is executed when the function SetActiveValue is called from a FACE script in the interface, or FmSetActiveValue is called from the application. The Set script is usually in charge of specifying a new value for the variable in the interface.

The Get script of an active value is executed when the function GetActiveValue is called from a FACE script in the interface, or FmGetActiveValue is called from the application. The Get script is usually in charge of reading the current value of the variable in the interface.

Note that an active value is not necessarily attached to the application. It can be used just to store scripts that will be triggered by events in the interface or by the application. If several widgets have an active value with the same name, it is possible to trigger their scripts selectively or collectively, just as it is possible to trigger the Get or Set script of all the active values associated to a given widget with one call. Refer to the description of the functions [Fm]GetActiveValue and [Fm]SetActiveValue.

Another use of active values in XFaceMaker is for the definition of new resources and callbacks when creating a widget class.

The syntax of the FACE language has the following additional constructs:

```
<widget>:@<avname>
<widget>:$<avname>
<widget>:<avname>(arg, ...)
```

These expressions can be used in the same manner as resource references, but will read, write or call an active value associated with a given widget. More precisely:

v = <widget>:@<avname> fetches the address attached to the specified active value and associated with the specified widget (possibly allocating a memory location of size sizeof(XtArgVal) if the active value is automatic), calls the *get* script of the active value with the address passed as the $ argument, and returns the contents of the address. The type of the result is the type of the active value. If the active value does not exist, or if it is immediate, or if it is not attached and it is not automatic, a FACE error occurs.

v = <widget>:$<avname> fetches the address attached to the specified active value and associated with the specified widget, calls the *get* script of the active value with the address passed as the $ argument, and returns the address. The type of the result is the type of the active value if it is immediate, or Pointer otherwise. If the active value does not exist, a FACE error occurs.

<widget>:@<avname> = v fetches the address attached to the specified active value and associated with the specified widget (possibly allocating a memory location of size sizeof(XtArgVal) if the active value is automatic), converts v to the type of the active value if v is of type String, stores v as the contents of the active value address, and calls the *set* script of the active value with the address passed as the $ argument. If the active value does not exist, or if it is immediate, or if it is not attached and it is not automatic, or if v is a String which cannot be converted to the type of the active value, or if v is not a String and the type of v is not compatible with the type of the active value, a FACE error occurs.

<widget>:$<avname> = v changes the address attached to the specified active value and associated with the specified widget, and calls the *set* script of the active value with the new address passed as the $ argument. If the active value does not exist, or if it is immediate and the type of v is not compatible with the type of the active value, or if it is not immediate and the type of v is not compatible with the type Pointer, a FACE error occurs.

<widget>:<avname>(arg, ...) can be used like a function call, but calls an active value's *set* script instead of a function. The arguments are stored in a structure, starting from the second field of the structure: the first argument in field 1, the second argument in field 2, etc. The active value's *set* script is called with the address of the structure passed as the $ argument for an immediate active value, or in the @ argument otherwise. The active value's script can reference the arguments by de-

claring a structure with the appropriate number of fields (i.e. the number of arguments plus one) and storing $ or @ in a variable typed with the structure name. The active value script can return a value by storing it in the first field of the structure.

Note that all fields of the argument structure must have the size of a FACE word, which means that you can declare the structure in FACE but you cannot use a structure declared in C with fields of size different from sizeof(XtArgVal).

7.3 Constants

XFaceMaker includes all the OSF/Motif constants defined in Xm/Xm.h, but not the definitions of resource names, classes and types. Thus, constants such as XmATTACH_WIDGET or XmDIALOG_OK_BUTTON are included, but not resources such as XmNwidth. Predefined OSF/Motif constants are of type “Int” or “String”.

7.4 Type Casters

The set of predefined type casters in FACE is extended to all the resource types known by the XFaceMaker resource editor.

7.5 Declaring Enumerated Resource Types

Enumerated resource type declarations in FACE are useful mainly in the context of widget class creation with XFaceMaker. As discussed above, a statement such as

```
type Tool = hammer, screwdriver, wrench, saw
```

declares a new type caster Tool, defines four constants, and a new type converter is registered in the X Toolkit so that the strings "hammer", "screwdriver", "wrench", "saw" can be converted to the corresponding constants.

In the context of widget class creation, the new resource type is also registered in XFaceMaker using the function FmAddEnumeratedType so that XFaceMaker pops the Enumeration box with the appropriate values when a resource of the new type is edited.

If the script is saved in a C file as a part of a new widget class, then the public header file of the class will contain C constants for the values of the enumeration, the C file of the class will contain a resource converter for the new resource type, and the function FmAdd*Name*WidgetClass will contain calls to register the new resource converter and to register the new type in XFaceMaker by calling FmAddEnumeratedType.

In short, if an enumerated type is declared in that way, XFaceMaker does everything you need to use the new resource type.

7.6 Structures

In XFaceMaker, all the structures defined in Xm/Xm.h are pre-registered with a name built as follows:

- if a pointer type is defined for the structure, then this type is used;
- otherwise, the FACE structure name is the name of the C structure type with the Struct postfix removed

For example, the callback structure for a push button widget is declared as XmPushButtonCallback in FACE while the text block structure used by the text widget is called XmTextBlock. The FACE types of the fields of pre-registered structures are derived from the C types:

- all integer scalar types are declared as Int
- Booleans are declared as Boolean
- character strings are declared as String
- other pointers are declared as Pointer
- other types are declared with their C names

7.7 Declaring Application Functions in XFaceMaker

When application functions are declared with the function keyword, and the same function is used in several callbacks of the same widget, it is sufficient to declare the function in the first callback listed in the Resources box, Callbacks window. An alternative option for a function that is called many times is to make the declaration in an xfmCreateCallback script which will be parsed when the interface is created.

To test scripts which call application functions using XFaceMaker's *Try* mode, it is often necessary to specify the return value for the function in simulation mode:

```
function Int my_fun(Widget, Int) = 23;
```

Another possibility is to incorporate application functions into XFaceMaker, using the functions FmAttachFunction and FmCallEditor to build a new XFaceMaker.

To test application functions in *interpreted* mode, you should, in your main program, attach the function, using the application function FmAttachFunction. If, in your interface, you have declared your application functions using the function keyword, and you are executing your application in interpreted mode, any warning messages that would normally be generated if the functions are not attached are suppressed.

7.8 Generating C-Code in XFaceMaker

XFaceMaker generates function calls for all operations on arrays. These functions are contained in the libFm_c.a library, which must be linked with the application if arrays are used. They are private and should not be used directly from the C-code of an application. When FACE scripts are saved in C-code, a FACE statement

```
return(x);
```

is translated into a C statement

```
return(x);
```

whereas return(); is translated into return;

If return is called outside a FACE function, it should always be called without arguments because the C functions corresponding to toplevel scripts are declared as void in the C files generated by XFaceMaker. Inside of functions, however, return statements should always be called with one argument, even if the function is declared as returning None, because FACE functions are translated as C functions returning an int.

7.8.1 Global FACE Functions

FACE functions can also be declared as global:

```
global function None Hello(String message)
{
        printf(“Hello, %s\n”, message);
    }
```

Global FACE functions differ from non-global functions only in the way their are output in the generated C code. Local FACE functions (i.e. FACE functions not declared with the global keyword) are output as static C functions, and are output in a C file only if they are actually called in the corresponding .fm file.

When a local function is called in several .fm files, the definition of the function will be replicated in all corresponding .c files. Global FACE functions, on the other hand, are output only in the C file corresponding to the .fm file where they are defined, as extern C functions. When a global function is called from another .fm file, it must be declared like this:

```
    global function None Hello(String message);
```

As for local functions, it is sufficient to declare a global FACE function once in a .fm file, provided that the declaration appears before all the calls. When a global function is called from a .fm file but not defined in this file, XFaceMaker outputs an extern declaration for the function.

7.8.2 Global Variables

In FACE scripts, global variables can be declared extern as in:

```
    extern global Int scale_value;
```

A variable declared `extern global' will be generated in the C code asa global extern variable:

```
    extern FaceVariable scale_value;
```

the first time it is used in a .fm file for which the C code is being generated.

The variable must be declared global without the `extern' in one and only one script:

```
    global Int scale_value;
```

This will produce the variable definition before the script in the C code:

```
    FaceVariable scale_value;
```

Using the `extern' modifier lets you use a global variable in several .fm files without having multiple definitions of the variable.

NOTE: If a variable is declared as `global' and not `extern global' in a file whose C code is generated separately from any other file referring to the variable, the definition of the variable is generated in the C code ONLY if the variable is USED in the file.

For example:

in A.fm:

```
......
activateCallback =
global Int scale_value;
.....
```

in B.fm:

```
....
activateCallback =
          extern global Int scale_value;
     .....
```

if A.fm and B.fm are compiled separately, e.g. with

```
xfm -compile A.fm
xfm -compile B.fm
```

the `extern global' declaration will not be known in A.fm and `scale_value', which is not used (only declared) in A.fm will not be declared in the A.c file.

If, however, the two files are compiled jointly, e.g. with

```
xfm -compile A.fm B.fm
```

then, `scale_value' will be declard in A .c with:

```
FaceVariable scale_value;
```

and in B.c with:

```
extern FaceVariable scale_value;
```

just as if the script in A.fm had actually used the global variable.

CHAPTER 8: Memory Management

This chapter describes how XFaceMaker handles memory allocation when interpreting scripts and how it handles the setting or fetching of resources.

During script execution, XFaceMaker does not implicitly copy buffers nor does it create memory buffers with malloc, but Motif does. The programmer is responsible for freeing memory in scripts. Therefore, it is important to know which resources, when accessed, provide their direct contents or a copy thereof. If they provide a copy, the corresponding buffer should be freed when no longer needed in order to keep memory usage within reasonable limits while running an application.

8.1 Accessing Resources

When the current status of a resource is required, XFaceMaker calls XtGetValues, and stores the result in a word. All FACE variables have the same size, that of a word. In certain cases, the Motif Toolkit allocates memory for values returned by XtGetValues. Normally, values returned by XtGetValues are never freed, but in the following cases you should free the values returned:

- when you retrieve the value of a compound string, such as labelString. It should be freed with StringFree in FACE or XmStringFree in the application.
- when you retrieve the value of the XmText or TextField value resource, which is of type String and not XmString, you should use free.

For example in a line such as:

```
buffer = XmText:value
```

a pointer to a *copy* of the widget's contents is returned in buffer. This should be freed with free when it is no longer needed.

Similarly, when a resource is set, XFaceMaker calls XtConvert if necessary, stores the result in a word, then calls XtSetValues. For the same widgets and resources as before, Motif makes a copy of the value specified for the resource; hence, the original buffer provided can be freed. For example:

```
parent.XmText:value = buffer
```

The contents of buffer is copied into the widget. buffer should be freed afterwards if it is not used elsewhere, and does not point to statically allocated memory.

In general, any resource of type XmString is copied into a new buffer whenever it is accessed with XtGetValues. The buffer thus created should be freed when no longer needed, using StringFree. Conversely, when an XmString resource is changed using XtSetValues, the widget keeps a copy of the XmString provided. Thus, the string provided can be freed after the XtSetValues.

Depending on your version of Motif however, there may be some XmString resources that do *not* make a copy when queried; the corresponding strings should not be freed. Copies are not returned for resources which hold XmStringTables. Therefore, they should *never* be freed. For other resources, you should refer to your Motif documentation to know if they are handled directly or through a copy, although the documentation is not always very specific.

8.2 Literal Strings

When literal strings are used in a script, the string is created by the XFaceMaker parser the first time it is encountered. The *same* string is used each time it is encountered by the FACE interpreter. For example:

```
self:label = "hello"
```

You should never attempt to free such a string. It will be freed when the widget is destroyed. If you want a copy of the string, create it using the function NewString, i.e., XtNewString, or the equivalent: malloc then strcpy.

If you repeatedly assign a string, some amount of memory will be lost on each XtSetValues of the XmString as explained above. Therefore, you should do the conversion explicitly if significant amounts of memory are involved:

```
str = CreateXmString("changing string");
 self:label = str;
 StringFree(str);
```

8.3 Eval Scripts

When using Eval scripts, for example in translations, you should always *name* the scripts. Each time it parses an unnamed Eval script, XFaceMaker allocates a new structure in which to store the result of the parsing. If it encounters a *named* Eval script, however, XFaceMaker creates the structure only once, then re-uses the already parsed script. The name identifies the script for XFaceMaker.

8.4 Returns from Function Calls

When calling functions from a FACE script or the application, you should refer to the function documentation to know what the function returns and, as a consequence, whether or not buffers should be freed. For example,(Xm)TextGetSelection returns a *copy* of the primary selection. However, any function which returns an XmString returns a pointer to a *copy*, which should be freed using StringFree when no longer needed, whereas functions returning an XmStringTable do *not* return a copy, and the pointer returned *should not* be freed.

8.5 Creating and Destroying Widgets

A further important case of memory usage, although not strictly related to FACE, is widget creation and destruction. Because of various bugs in the X and Motif toolkits, the memory allocated when creating a widget instance *may not* be completely freed by a call to DestroyWidget (XtDestroyWidget). Destruction of widgets should therefore be minimized; it is, in fact, much faster to unmap and map a widget when required.

CHAPTER 9: Structures

This chapter describes some of the structures registered or defined in XFaceMaker. All the Motif callback structures are registered. The table below lists the names of structures in FACE corresponding to the Motif structure names.

Two structures are defined in FACE for data transfer control in Drag and Drop: the FmConvertProcStruct and the FmTransferProcStruct. These provide callback information regarding the data being transferred.

The variable names you should use to access some of the fields in the XEvent structures for ClientMessage events and KeymapNotify events are provided in section 9.2.

Table 9.1: Structures

Fm Name	Xm Name
XmAnyCallback	XmAnyCallbackStruct
XmArrowButtonCallback	XmArrowButtonCallbackStruct
XmDrawingAreaCallback	XmDrawingAreaCallbackStruct
XmDrawnButtonCallback	XmDrawnButtonCallbackStruct
XmPushButtonCallback	XmPushButtonCallbackStruct
XmRowColumnCallback	XmRowColumnCallbackStruct
XmScrollBarCallback	XmScrollBarCallbackStruct
XmToggleButtonCallback	XmToggleButtonCallbackStruct
XmListCallback	XmListCallbackStruct
XmSelectionBoxCallback	XmSelectionBoxCallbackStruct
XmCommandCallback	XmCommandCallbackStruct
XmFileSelectionBoxCallback	XmFileSelectionBoxCallbackStruct
XmScaleCallback	XmScaleCallbackStruct
XmTextBlock	XmTextBlock
XmTextVerifyPtr	XmTextVerifyPtr
XmTextBlockWcs	XmTextBlockWcs
XmTextVerifyPtrWcs	XmTextVerifyPtrWcs
XmTraverseObscuredCallback	XmTraverseObscuredCallbackStruct
XmSecondaryResourceData	XmSecondaryResourceData
XmAnyICCCallback	XmAnyICCCallback
XmTopLevelEnterCallback	XmTopLevelEnterCallback
XmTopLevelLeaveCallback	XmTopLevelLeaveCallback
XmDropSiteEnterCallback	XmDropSiteEnterCallback
XmDropSiteLeaveCallback	XmDropSiteLeaveCallback
XmDragMotionCallback	XmDragMotionCallback
XmOperationChangedCallback	XmOperationChangedCallback

Table 9.1: Structures

Fm Name	Xm Name
XmDropStartCallback	XmDropStartCallback
XmDropFinishCallback	XmDragProcCallback
XmDragDropFinishCallback	XmDropFinishCallback
XmDragProcCallback	XmDragDropFinishCallback
XmDropProcCallback	XmDropProcCallback
XmDropSiteVisuals	XmDropSiteVisuals

9.1 Conversion and Transfer Structures

The drag source and drop site active value scripts for the exported and imported targets can (and sometimes must) access additional information about the data being transferred. This information is passed to the active values scripts as the address of a structure. The structure fields can be accessed in FACE using the @n syntax. They can also be accessed as FACE structure fields using the -> operator.

The $ argument of a drag source's active value for an exported target points to an **FmConvertProcStruct**:

```
typedef struct _FmConvertProcStruct {
    XtPointer      value;
    String         type;
    int            length;
    int            format;
    Widget         drag_context;
    String         selection;
    String         target;
    XtPointer      client_data;
    unsigned long  max_length;
    XtRequestId    request_id;
} FmConvertProcStruct;
```

The fields of this structure are derived from the values passed to the convertProc of the drag context, which is provided by XFaceMaker. The Atoms are changed to strings, and some return values are initialized to default values. Refer to the X Toolkit Intrinsics manual, section 11.5.2.1, for a complete description of selection conversion procedure arguments.

value is where the active value's *get* script must store the value to transmit to the drop site. If the value is immediate (i.e. it is an integer or a one-word value), then it must be stored directly in the value field, and the length field must be left to its initial value of 0. For a string or a pointer then, the pointer must be stored in the value field. For targets STRING, TEXT and COMPOUND_TEXT, the length field is updated automatically to the length of the string (which must be null-terminated). For other targets for which a pointer is stored, the length field must be updated explicitly to the size of the data.

type is where the type of the data must be stored. This field is initialized to a default value which is taken from the table of "generally accepted" atoms listed in the ICCCM, section 2.6.2. For the commonly used targets STRING and COMPOUND_TEXT, the type is initialized to the target name. For the TEXT target, the type is initialized to TEXT, but it is changed to STRING if left alone. For targets not in this table, the type field is initialized to the target name.

length is where the length of the converted data must be stored if the data is a pointer (and not a string). The length is in units defined by format.

format is where the format of the converted data must be stored. This can be 8, 16, or 32. This field is initialized to 8 if type is STRING, TEXT and COMPOUND_TEXT, and to 32 otherwise.

drag_context is the XmDragContext widget returned by XmDragStart.

selection is the name of the selection used by the Motif drag and drop mechanism: _MOTIF_DROP.

target is the name of the selection target. It is the same as the active value name.

client_data is the value of the XmNclientData resource passed to XmStart-Drag, if any.

max_length, *request_id* These are the arguments passed to the conversion procedure for incremental transfers (refer to the X Toolkit Intrinsics documentation).

The $ argument of a drop site's active value for an imported target points to an **FmTransferProc**:

```
typedef struct _FmTransferProc {
            XtPointer     value;
            String        type;
            int           length;
            int           format;
            Widget        drop_transfer;
            String        selection;
            String        target;
            XtPointer     client_data;
} FmTransferProc;
```

The fields of this structure are derived from the values passed to the transferProc of the drop transfer object, which is provided by XFaceMaker. The Atoms are changed to strings. Refer to the X Toolkit Intrinsics manual, section 11.5.2.2, for a complete description of selection callback procedure arguments.

value is where the active value's *set* script can read the value transmitted from the drag source. If the value is immediate (i.e. it is an integer or a one-word value), then it is stored directly in the value field, and the script must NOT free any memory. For a string or a pointer, then the pointer is stored in the value field, and the script can free the memory when no longer needed.

type contains the type of the data.

length contains the length of the converted data.

format contains the format of the converted data.

drop_transfer contains the XmDropTransfer widget returned by XmDropTransferStart.

selection contains the name of the selection used by the Motif drag and drop mechanism.

target is the name of the selection target.

client_data contains the value of the XmNclientData resource passed to XmDropTransferStart, if any. If XmDropTransferStart was not used directly but called automatically by the drop site's XmNdropProc, the client_data is the XmNdropProc's call data (of type XmDropProcCallback).

9.2 XEvent Structures

The Client_Message fields of the ClientMessageEvent structure can be accessed in FACE through intermediate variables as listed below.

	in FACE
char b[0]	data_b0
char b[1]	data_b1
.	
.	
char b[19]	data_b19
short s[0]	data_s0
short s[1]	data_s1
.	
.	
short s[9]	data_s9
long l[0]	data_l0
long l[1]	data_l1
.	
.	
long l[4]	data_l4

Similarly, in KeymapNotify events, the key_vector fields are accessed as follows:

	in FACE
char key_vector[0]	key_vector0
char key_vector[1]	key_vector1
.	
.	
char key_vector[31]	key_vector31

CHAPTER 10: The Genappli Utility

This chapter describes in detail the program **genappli**. This program is used in XFaceMaker to generate application files, C++ class source files, and Windows source files. It could also be used independently of XFaceMaker.

The genappli program combines two input files, a *description file* and a *pattern file*, to produce one output file.

The description file contains information specific to the interface for which files are being generated, and is produced by XFaceMaker. For example, the description file contains the names of the interface modules, the types of their top-level widgets, the active values defined, etc.

The pattern file describes the code that will be generated and is specific to the kind of file being produced. For example, there is one pattern file for the "main" of an XFaceMaker application, and another pattern file for the Makefile. The pattern file contains patterns that will be matched against the information contained in the description file. The pattern file can also contain plain code that will simply be copied to the output file.

The output of genappli can optionally contain a copy of the input patterns. In that case, the output file can in turn be used as an input pattern file. If code has been added to the output file, it is preserved the next time the file is generated.

10.1 Description File Syntax

The description file contains *pattern descriptions* of the form:

```
<pattern-name> <object-name>
    <attribute1> = <value1>
    <attribute2> = <value2>
    ...
;
```

All symbols (pattern names, object names, attribute names and attribute values) are arbitrary strings. Symbols do not have any predefined semantics, except for the single keyword children explained later. The following characters act as delimiters, as well as blanks, tabs and newlines:

```
~ ! @ # $ % ^ & * () + ` - = {} | [] \ : " ; ' <> ? , . /
```

Symbols containing special characters must be quoted using matching sequences of any number of back-quote and quote characters, e.g.: ‘‘‘Hello’’’

An attribute value can be either a single string or a list enclosed in parentheses:

```
function Hello
    RETTYPE = void
    NUMARGS = 2
    ARGNAMES = ( indent msg )
    ARGTYPES = ( int ‘char *’ )
;
```

(The convention is to use upper-case characters for attribute names, to make them more visible in pattern files, but this is not mandatory).

Pattern descriptions can contain sub-patterns, which are defined using the special attribute name children:

```
function Hello
    RETTYPE = void
    NUMARGS = 2genappli
    children = (
        argument indent
            TYPE = int
        ;
        argument msg
            TYPE = ‘char *’
        ;
    )
;
```

Sub-patterns can nest at any depth. There may be more than one children attribute, in which case the values of all children attributes are concatenated.

10.2 Pattern Files

Pattern files contain plain code that is copied to the output file, and pattern definitions. Pattern definitions must be defined inside comments for the target language, and are identified by special character sequences depending on the language. For example, if the output file is a C source file, pattern lines begin with “/*## “ and end with “ ##*/”, whereas if the output file is a Makefile, pattern lines begin with “####

“ and no special character sequence is necessary at the end of the line. The pattern line character sequences can be specified as command-line options. In the examples below, we will use the C syntax.

Patterns are enclosed within lines of the form:

```
/*## begin pattern <pattern-name> ##*/
...
/*## end pattern <pattern-name> ##*/
```

Pattern lines are broken into tokens separated by blanks or special characters which are the same as for description files. Tokens can also be quoted in the same manner as in description files, using back-quote/quote sequence pairs. Contiguous tokens can be separated by quoting one of the tokens, or both.

The begin pattern line can contain an optional conditional expression, in one of two forms:

```
/*## begin pattern <pattern-name> <left> <op> <right> ##*/
/*## begin pattern <pattern-name> [ <expression> ] ##*/
```

In the first form, <op> can be one of = (equal), # (not equal), [(is contained in) or] (contains). For the “contains/is contained in” operators, one of the values must be the name of an attribute of which the value is a list: the operator tests if the other value is contained in the list. This form of expressions is obsolete.

The second form of expressions (enclosed in square brackets) allows more complex conditions using a C-like syntax. Attributes are replaced by their values. The C operators &&, ||, <=, <, >=, >, ==, !=, +, -, *, /, !, -, +, (and)can be used with the same semantics as in C except for == and != which can be used to compare character strings as well as integers. Arithmetic operators treat non-numeric character strings as 0. In addition, the following operators are available:

s1 { s2 : If s2 is a string, if string s2 contains s1, returns the position (1-N) of s1 within s2. Else, returns 0. If s2 is a multiple element list, test is s1 is a member of the list and returns its position.

s1 } s2 = s2 { s1

s [i If s is a multiple element list, returns string number i (starting from 1). Else, returns character number i

s1 left s2 : If s1 contains s2, returns the sub-string to the left of s2 within s1. Else, returns s1.

s1 left i : Returns the first i characters of s1.

s1 right s2: If s1 contains s2, returns the sub-string to the right of s2 within s1. Else, returns an empty string.

s1 right i : Returns the i last characters of s1.

upper s: converts string s to upper-case.

lower s: converts string s to lower-case.

length s: returns the length of string s.

cstring s: returns string s with its special characters escaped so that it can be used in a C program.

defined s : Returns 1 if s is a defined value or variable, i.e. in an expression, it would be substituted.

value s : Explicitly substitutes value s. Use this operator to obtain the values of variables whose names were returned by “allvalues”.

For each pattern found in the pattern file, instancing takes place as follows.

- If the -c option is not set, the pattern definition is first copied to the output file.
- The pattern definition is instantiated using each pattern description with the same name found in the description file. If the pattern definition contains a conditional expression, the expression is evaluated and the pattern is instantiated only if the result is non-zero.
- The pattern file is searched for an existing instance of the pattern in the pattern file. If a pattern instance is found, the user sections in it are recorded. (See the explanation on the begin user ... end user commands below.)
- The pattern is read line by line. For each line, the tokens which are equal to attribute names in the pattern description are replaced with the attribute values. Then, the line is output.
- User sections in the pattern are filled with the contents of the user sections of the existing pattern instance, if any.
- Line beginning with command keywords are processed. See the list of commands below.
- Pattern instances are enclosed within lines of the form:

  ```
  /*## begin <instance-name> ##*/
  ...
  /*## end <instance-name> ##*/
  ```

Attributes of the parent pattern can be used by prefixing ‘‘__’’ (two underscore characters) to the attribute name. Adding several ‘‘__’’ prefixes goes up several levels in the pattern hierarchy.

A special construct allows you to iterate through lists of values. If a pattern line contains:

```
/*## ... $$ATTR1 ATTR2 ... $<sep>$ ... ##*/
```

then the values of the attributes ATTR1, ATTR2, etc. (which must be lists) are replaced iteratively for all list positions, with the separator string <sep> inserted at the end of each “row”. The list iterator character can be specified through a command-line option.

Patterns can contain commands identified by keywords which must occur at the beginning of the pattern line. Here is a list of the pattern commands.

```
/*## begin user ##*/
...
/*## end user ##*/
```

These commands identify a code section that is to be filled with code added by the user in user sections of an existing instance of the pattern in the pattern file.

```
/*## begin pattern <pattern-name> ##*/
/*## begin pattern <pattern-name> <left> <op> <value> ##*/
/*## begin pattern <pattern-name> [ <expression> ] ##*/
...
/*## end pattern <pattern-name> ##*/
```

Patterns can contain sub-patterns, which correspond to sub-pattern descriptions. Sub-pattern replacement is done as if the pattern file contained only the enclosing pattern and the description file contained the sub-pattern description as a top-level object. The optional expression allows you to control the conditional instanciation as for a top-level pattern.

```
/*## recursive pattern <pattern-name> ##*/
/*## recursive pattern <pattern-name> [ <expression> ] ##*/
```

Recursive patterns are a special kind of sub-patterns. This command can occur only inside the definition of the named pattern. The effect is as if the whole definition of the named pattern were included in place of the recursive pattern command. A conditional expression is allowed, but only in the second form.

/*## call pattern <name> [<condition>] ##*/

Call a pattern defined with the instruction “define pattern”. The effect is like “begin pattern ... end pattern” but the pattern called can be defined elsewhere.

/*## call define <name> ##*/

Call a pattern, but use the same values as for the enclosing pattern.

/*## call command <cmd> <arg> ...##*/

Execute a Shell command from genappli.

/*## call include <file>##*/

Include a pattern file in another pattern file. The included file should contain pattern lines, and no user code (code copied directly without processing).

/*## set local <variable> = <expression> ##*/

Set a local variable. The value “variable” is added to the current object, just as if it had been defined in the .gen file. “expression” can be:

- a succession of words - the words will be substituted as in a standard code line, then the substituted values will be placed in a list associated to the variable.
- an expression within “[]” - the expression is evaluated and the result applied to the value.

/*## set global <variable> = <value> ##*/

Set a global variable. Same action as for local, but the value applies to all the patterns in the pattern file, regardless of the object type.

```
/*## begin if [ <expression ] ##*/
...
/*## elseif [ <expression> ] ##*/
...
/*## ifelse ##*/
...
/*## end if ##*/
```

Pattern lines enclosed within the branches of the if command are instantiated only if the corresponding condition is non-zero. “If” commands can nest at any level.

```
/*## begin oneline ##*/
...
/*## end oneline ##*/
```

All pattern lines inside a “oneline” section are output on the same line. “Oneline” sections can nest: only the end of the top-level “oneline” section will start a new line.

/*## forcenewline ##*/

Force a newline character to be output inside a “oneline” section.

10.2.1 Pattern File Objects

The main objects used in the application generation pattern files are listed in this section. Attributes that have a list value are marked with “()”. The value of other attributes is a character string.

10.2.1.1 Interface Description

Used to describe a project - these objects are used once in a project.

NAME The application name (the name of the main **.fm** file)

LCNAME Application name with leading lower-case

UCNAME Application name with leading upper-case

CREATENAME Creation name (value of **-cname** option)

TARGETMACH Name of the target operating system for compilation and link - used in Makefile pattern

TARGETMACHFILE Name of the configuration file Mk-<machine> that should be used to compile and link - used in Makefile pattern

XFMNAME Name of the XFaceMaker executable used to generate the application files

CLASS Class of the interface’s tope level object

NUMAVS Number of global active values

NUMNAMES Number of **.fm** files in the interface

COPTIONS () C-code generation options (**-cflags** option)

NAMES () Base names of the **.fm** files in the interface

PROJECTFILE Name of the project file

FILES () Names of the **.fm** files

LCNAMES () Names of the **.fm** files (lower-case)

BASENAMES () Names of the **.fm** files (upper-case)

CREATENAMES ()	Creation names
TOPOBJS ()	Types of toplevel objects
MAPPED ()	Mapped flag for topelevel objects
PARENTPATHS ()	Paths, in the Motif hierarchy, of toplevel objects
NUMUCLASSES	Number of *dynamic* user classes added to *this* XFaceMaker
UCLASSPREFIX	User class prefix (**-cprefix** option)
UCLASSNAMES ()	User *(dynamic)* class names
UCLASSFILES ()	File names for user *(dynamic)* classes
UCLASSPATHS ()	Full path names for user *(dynamic)* classes
NUMACLASSES	Number of classes added *(statically)* to this XFace-Maker
ACLASSNAMES ()	Names of classes added *(statically)* to this XFace-Maker
NUMUSEDACLASSES	Number of added classes used in the in-terface
USEDACLASSNAMES ()	Names of added classes used in the in-terface

10.2.1.2 Module Description

Used to describe each **.fm** file in the project.

NAME	Base names of the **.fm** file
FILE	Name of the **.fm** file
LCNAME	Name of the **.fm** file (lower-case)
BASENAME	Name of the **.fm** file (upper-case)
CREATENAME	Creation name
INDEX	Index of this **.fm** file in the project
TOPOBJ	Type of toplevel object
MAPPED	Mapped flag for topelevel object
PARENTPATH	Path, in the Motif hierarchy, of the toplevel object

10.2.1.3 Active Value Description

Used to describe each toplevel global active value.

CLASSNAME	C++ class name
NAME	Name of the active value
UCNAME	Name of the active value with leading upper-case character.
TYPE	Active value type in the FACE sense
CTYPE	Active value type in the C sense (the type in C-code generated)
USAGE	Active value usage: FACE, class, cxx ....
STORAGE	Active value storage: none, object, ...
SCOPE	Active value scope in C++ : private, public, ...
IMMEDIATE	true if immediate active value, false otherwise
AUTOMATIC	true if automatic allocation, false otherwise
GENFUN	In C++, tru if automatic function generation was requested
VALUE	Initial (default) value in string format

10.2.1.4 Resource Description

Used to describe each resource of the toplevel object

NAME	Name of the resource
UCNAME	Name of the resource with leading upper-case character
TYPE	Resource type in the FACE sense
CTYPE	Resource type in the C sense (the type in C-code generated)
VALUE	Resource value in string format

10.3 Genappli Options

The genappli program accepts the following command-line options and arguments:

```
genappli [-p <pattern-file> [-a <application-file>]
[    -o <output-file>] [-b <begin-patt-string>]
     [-e <end-patt-string>] [-l <list-delimiter-string>]
     [-m <output-file-fopen-mode>] [-c] [-q]
     [<pattern-file>] [<application-file>] [<output-file>]\n",
```

-p <pattern-file>: specifies the pattern file to use.

-a <application-file>: specifies the application pattern description file to use.

-o <output-file>: specifies the output file.

-b <begin-patt-string>: specifies the character string which identifies the beginning of pattern lines.

-e <end-patt-string>: specifies the character string which identifies the end of pattern lines.

-l <list-delimiter-string>: specifies the character used to delimit list iterations.

-m <output-file-fopen-mode>: mode passed to “fopen” to open the output file.

-c: (clean) if set, do not output pattern definitions in output file.

-q: (quiet) do not print messages on the standard output.

<pattern-file>: specifies the pattern file to use.

<application-file>: specifies the application pattern description file to use.

<output-file>: specifies the output file.

10.4 Genappli Error Messages

When genappli encounters an unrecoverable error, it displays the line number where the error occurred and an error message. You should first determine which pattern files you are working with, then examine the faulty file, correct the error, then re-issue the command to generate your files.

Table 10.1: Application Generation Error Messages

Error Constant	Value	Meaning
ERR_NO_TMPL	100	no patterns found in file
ERR_NO_NAME	101	no name found for pattern
ERR_NO_KEY	102	no key
ERR_NO_START	103	encountering end of pattern without a start
ERR_NEST_USER	104	nested user parts not allowed
ERR_NO_USER	105	encountering end of user part without a start
ERR_BAD_LIST	106	incorrect list syntax
ERR_LIST_MISMATCH	107	list mismatch
ERR_PARSE_ERROR	108	parse error
ERR_CANT_READ_TMPL	109	cannot read pattern file
ERR_CANT_READ_DESC	110	cannot read description file
ERR_USAGE	111	incorrect program arguments
ERR_BAD_COND	112	incorrect condition syntax
ERR_BAD_IF	113	incorrect `if' or `while' syntax
ERR_IFLESS_ENDIF	114	if-less end-if (or while-less end-while)
ERR_IFLESS_ELSE	115	if-less ifelse
ERR_IFLESS_ELSEIF	116	if-less elseif
ERR_BAD_PATT_CALL	117	incorrect pattern call
ERR_BAD_DEPTH	118	incorrect depth specification in variable assignment
ERR_BAD_SUB_PATT	119	incorrect sub-pattern
ERR_STOP	120	generation stopped
ERR_NO_WHILE	121	end while without a begin
ERR_VARIABLE_STACK_BOT	122	variable stack bottom reached (too many `__'s)
ERR_FATAL_SIGNAL	123	fatal signal caught

To determine which pattern files you are using, proceed as follows:

1. If the ***Preserve User Code*** toggle is set, and you have already saved your application files once, check these. There is a good probability that you have made a mistake when adding your code.
2. If the toggle is not set, or if you have not yet generated any application files for the current interface, examine the pattern files that are in the GenAppli directory under:
    1. The current XFaceMaker directory.
    2. Your $HOME directory.
    3. The directory $NSLHOME/lib/xfm.

CHAPTER 11: New Widget Classes

XFaceMaker enables you to define entirely new classes of widgets, which inherit much of the behavior of existing classes, but to which you can add resources and/or functions for your own special purposes. You can create the new class:

- *statically* by generating a C-code file containing the source code for the new class.
- *dynamically* by creating the new class while running XFaceMaker. This enables you to instantiate and test the new class immediately.

This chapter describes the resources and methods for new widget classes. Generating a new class is described in the Chapter entitled “New Widget Classes” in the *XFace-Maker User’s Guide*.

11.1 Widget Class Resources

A new widget class generated by XFaceMaker has two kinds of resources:

- superclass resources—resources inherited from the model object of the class, with different default values.
- new resources—resources defined through the active value mechanism.

11.1.1 Superclass Resources

When XFaceMaker generates a class definition file, every resource that was specified for the model object of the new widget class is included. The resource definition is a copy of the original resource definition in the superclass, except for the default_type and default_addr fields which are set according to the value specified for the object.

For example, suppose you create an XmPushButton, set the background resource to "red", and save it as a new class, “PushButton”. This new widget class has a background resource with the same type, size, representation and offset as the original **Core** background resource, but with a default value of "red". This value can still be overridden by any of the usual means, such as resource database files or widget creation arguments.

The callbacks of the model object are not added to the resource list, but set in the Initialize method using XtAddCallback. The translations resource is also handled in a special way.

- If the translation table begins with #replace (or with a character other than #), the translation table is simply stored in the widget classes tm_table field which defines the default translation table for the class.
- If the table begins with #override or #augment, the translation table is installed in the Initialize method by calling XtOverrideTranslations or XtAugmentTranslations. This allows the widget designer to override or augment the superclass’ default translations, which is not possible otherwise. The default translations of a class are either completely inherited or completely re-defined.

Note that if #override or #augment are specified in the model's translations, it will not be possible to re-define the specified translations in a particular instance of the class. If you want to be able to re–define the instance's translations, you must specify #replace (or nothing), but in that case you will have to copy all the translations of the superclass.

If no translations are defined for the model object, the translations of the new class are inherited from the superclass. To define a new class with no translations, use #replace followed by an empty translation table.

11.1.2 Active Value Resources

When XFaceMaker generates a class definition file, every active value resource that was specified for the model object of the new widget class is included, except in the following cases.

- The following active value names are reserved and are used to define the corresponding methods of the widget class:

AcceptFocus	ChangeManaged	ClassInitialize	DeleteChild
Destroy	DisplayAccelerator	Expose	GeometryManager
Initialize	InsertChild	QueryGeometry	Realize
Resize	SetValuesAlmost		

This is explained further in Section 11.2 .

- If there is already a superclass resource with the same name as the active value, then the resource is not redefined, but the Set and Get scripts will still be called by the *SetValues* and *GetValuesHook* methods as explained in Sections 11.2.4 and 11.2.5. This can be used to define additional side-effects for an existing resource of the superclass.

The type of a new resource can be specified in the type field of the Active Values editing box. The resource type is a FACE type. To define a new resource type, you can declare it in the Set script using the type keyword.

Remember that the type of your new resource will be used for converting strings to values suitable for the resource. Here is what you will have to do, depending on the type of your new resource.

- If the new resource can be represented by an existing Xt or Motif type, you must use the representation type string corresponding to the existing type. For example, if your resource is an integer value, you must give the type Int, which is predefined by Xt and for which a String to Int converter exists. In this case, you don't have to do anything else.

- If the new resource is *enumerated*, i.e. can take a fixed number of integer values, then XFaceMaker can generate all the functions necessary to handle the new type. All you have to do is give the names of the different values using the type keyword, which has been extended for that purpose. For example, you can put in the Set script for a new resource:

  ```
  type MyEnum = choice1, choice2, choice3;
  ```

  and enter MyEnum as the active value type.

  This defines a new enumerated type MyEnum, and registers automatically a resource converter from String to MyEnum. The values are defined as FACE constants: *Prefix*CHOICE1, *Prefix*CHOICE2 and *Prefix*CHOICE3, where *Prefix* is the class prefix that has been specified in the Save Options box (or with the -cprefix option) if any, or Xfm otherwise. The values will also be output as C constants, with the same names as the FACE constants in the public header file, when the new class is saved as C code. Finally, the FmAddEnumeratedType function is called automatically so that XFaceMaker will pop the Enumeration box filled with the right values when the new resource is selected in the resource list of the Resources box of XFaceMaker.

- If the new resource is of a completely new data type that cannot be represented by any existing type, then you will have to provide a type converter which will be called by the X Toolkit to convert strings to the data type of your new resource. See the X Toolkit documentation for a complete description of type converters and the way they must be registered. Also, you might want to design a special editing box which will be popped by XFaceMaker when the new resource is selected in the resource list of the Resources box of XFaceMaker. See the section entitled "Editing a New Resource Box" in the *XFaceMaker User's Guide*.

Active values are translated into resources as follows:

- resource name is the name of the active value, which should start with a lower-case letter and follow the Xt Intrinsics conventions;
- resource class is the name of the active value with its first letter in upper-case;
- resource type is the type obtained as explained above, or otherwise XtRString;

- resource size is one long word (or four bytes), except if the resource type is an existing Xt or Motif type corresponding to a data type of a different size (for example, if the resource type is Boolean, then the resource size is one byte);
- resource offset is the offset of the memory location allocated in the widget structure for the resource;
- default address is 0 if no default value is specified for the resource, otherwise a pointer to the default value string;
- default type is XtRImmediate if no default value is specified for the resource, otherwise XtRString.

Default values can be specified in the Resource Edit box, like other resource values, i.e., arrows = XfmARROWS_LEFT, max = 10. Active values are also listed in the resource list, so you can choose the default value as you would for ordinary resources.

11.2 Widget Class Methods

A widget class method is a function that you can call to operate on an object. This section describes the various methods that can be generated for a new widget class, based on a model object definition.

There are fourteen reserved active value names, each corresponding to a method of the Core or Composite widget class. When one of these active values is defined for the model object, XFaceMaker generates a class method which calls the Set script of the active value. For every undefined active value, XFaceMaker replaces the method pointer by the appropriate XtInherit value, to indicate that the method must be inherited from the superclass.

The parameters passed to the methods by the X Toolkit are transmitted to the active value scripts in a structure whose address is passed as the $ argument. The structure passed is different for each method, and is explained below for each case. The argument names are those used in the X Toolkit Intrinsics manual by the X Consortium (McCormack89). The Class Method Arguments table summarizes the arguments passed to the method scripts.

For simple widget classes, most of the methods need not be defined— the inherited methods can usually be used. In most cases, you will only need to define the Set scripts for your new resources.

You can also initialize variables, declare functions or register converters in the Initialize method. Methods such as Realize, SetValuesAlmost, etc... are used less often, and should merely call C functions written by the application designer.

Defining inappropriate scripts (e.g. bad pointers or bad values given as resource values) for certain methods can confuse the X Toolkit and cause XFaceMaker to crash. In general, all the pitfalls of Xt programming in C can occur in FACE scripts. However, due to XFaceMaker's dual-process architecture, the design process may crash but the XFaceMaker process stays alive.

Only the methods of the Core and Composite classes can be defined using active values. Widget designers who wish to define other methods defined by a sub–class of Core or Composite must do so by calling a C function that sets the appropriate pointers in the class record. This function must be called in the ClassInitialize method.

11.2.1 The Class Method Arguments

Method	self	@0/ $	@1	@2	@3
ClassInitialize	---	---	---	---	
Initialize	*new*	request[1]	---	---	
Realize	*w*	value_mask	attributes	---	
Destroy	*w*	---	---	---	
Resize	*w*	x of w	y of w	width of w	height of w
Expose	*w*	event	region	---	
SetValues	*new*	value in new	value in current	current[2]	
SetValuesAlmost	*new*	old	request	reply	
GetValuesHook	*w*	value	---	---	
AcceptFocus	*w*	time	return value	---	
QueryGeometry	*w*	request	geometry_return	return value	
DisplayAccelerator	*w*	string	---	---	
GeometryManager	par*ent of w*	w	request	geometry_return	return value
ChangeManaged	*w*	---	---	---	
InsertChild	*parent of w*	w[3]	---	---	
DeleteChild	*parent of w*	w[4]	---	---	

Table 11.1: Class Method Arguments

1. @ is undefined, value is in $
2. Or 0 if called from Initialize.
3. @ is undefined, value is in $.
4. @ is undefined, value is in $.

11.2.2 The ClassInitialize Method

The ClassInitialize method can be used for doing one-time initializations for the widget class, such as registering resource converters. The default ClassInitialize method sets the Composite class part methods defined by the active values GeometryManager, ChangeManaged, InsertChild and DeleteChild if any, for sub–classes of Composite. If an active value with the name ClassInitialize exists for the model object of the class, its Set script is called in the class_initialize method, with a NULL argument. Therefore @ must not be referenced.

11.2.3 The Initialize Method

The Initialize method of the new widget class does the following:

- adds the callbacks defined for the model object;
- creates the child widgets of the new widget instance, according to the children of the model object if any exist;
- calls the creation callback for the new widget if one is defined;
- calls the Set scripts of all active value resources to initialize them (see Section 11.2.4);
- calls the Set script if an active value with the name Initialize exists for the model object of the class.

The CreateCallback or the Initialize Set script should be used to set the initial values for any resource whose default values cannot be set directly in the Resource Editor Box. However, these two are not strictly equivalent:

1. The CreateCallback script is called every time the model object is rebuilt; it cannot be used to initialize new resources of the class being defined, since these do not yet exist as such in the model object. They are still only Active Values when the model object is being rebuilt.
2. The Initialize Set script is called only after the new class is saved when instances of the class are being created. At this time, all the resources of the new class exist and are accessible.

For example, if you design a class which consists of a BulletinBoard and some children, and you want to set the defaultButton resource, you can do it in the CreateCallback: the resource exists in the BulletinBoard class. Moreover, by setting the resource in the CreateCallback you have the advantage of being able to check the behavior directly on the model object:

```
xfmCreateCallback = self:defaultButton = self.ok_button;
```

On the other hand, if, in your new widget class, you are defining a new resource called applyButton, whose type is a widget ID, you can set it only after the class has been created, i.e. in the Set script associated to the initialize active value. You cannot set it in the CreateCallback because the resource does not exist in the model object: XFaceMaker would produce an error message. The Set script of your initialize active value might be:

```
set_Initialize = self:applyButton = self.apply_button;
```

At the time this script is executed, both the apply_button widget and the applyButton resource exist.

11.2.4 The SetValues Method

The SetValues method handles changes in the new resources you specify as active values in the model object of the class. This method is called by the X Toolkit at every resource change during widget use; i.e., calls to XtSetValues. The **SetValues** method does the following:

- fetch the current value of the resource from the *current* widget argument.
- fetch the new value of the resource as specified by the call to XtSetValues from the *new* argument,
- call the Set script of the active value, with the call_data pointing to the following structure:

```
struct {
    XtArgVal new_value;
    XtArgVal current_value;
    XtArgVal current_widget;
}
```

Thus, in the Set script,

@ refers to the new resource value,

@1 refers to the old resource value,

@2 contains a pointer to the *current* widget.

The same sequence of operations is performed in the Initialize method, but in this case new_value is equal to current_value and the pointer is NULL; hence in the Set script @1 == @, and @2 == 0.

In the Initialize method, the Set scripts of *all* the new resources you have defined for your class are called, regardless of the creation arguments passed by the application to XtCreateWidget.

Calling the new resource's Set scripts both at Initialize and SetValues time ensures a homogenous behavior of the new resource: it is better if setting a resource at the creation of the widget and changing it by SetValues has the same effect. However, it is possible to differentiate the two cases by testing the @2 argument.

The SetValues procedure *always* calls the Set script of a resource, even if the resource value has not actually changed. You can test that a resource value has really changed using the arguments passed to the Set script in order to avoid performing any useless side-effects. A simple way to do this is to start the Set script with the following:

```
        if(@2 && @ == @1) return();
```

This ensures that the value is processed at initialization time (@2 == 0), and that the new value (@) is different from the old value (@1). The SetValues method always returns False, meaning "do not redisplay". Widget classes wishing to be redisplayed when a resource is changed should explicitly call a C function which clears the widget's window, causing a redisplay.

11.2.5 The GetValuesHook Method

The GetValuesHook procedure is called when XtGetValues is invoked on the widget. For every resource of the new widget class that was specified by the caller of XtGetValues, it performs the following:

- fetches the current value of the resource from its widget argument.
- calls the Get script of the active value, with the call_data pointing to the current value; thus in the Get script, @ refers to the current resource value.
- stores the value, which may have been modified by the Get script, into the address specified by the caller of XtGetValues.

Usually, you do not need to specify a Get script for your new resources, since side-effects when reading resource values are not frequent.

11.2.6 The Resize Method

If an active value with the name Resize exists for the model object of the class, its Set script is called in the class Resize method, with the following arguments:

@0 = x position of the widget,

@1 = y position of the widget,

@2 = width of the widget,

@3 = height of the widget,

If a Resize method is specified, the superclass's Resize is not called automatically by the Intrinsics. For this reason, a convenience function, SuperclassResize(w), is provided, which should be called in most cases in your Resize method.

11.2.7 The Expose Method

If an active value with the name Expose exists for the model object of the class, its Set script is called in the class Expose method, with the following arguments:

@0 = event parameter of the Expose method

@1 = region parameter of the Expose method

As for Resize, a convenience function, SuperclassExpose(w, event, region), is provided, which should be called in most cases in your Expose method:

```
SuperclassExpose(self, @, @1);
```

11.2.8 The Destroy Method

If an active value with the name Destroy exists for the model object of the class, its Set script is called in the class Destroy method, with a $ NULL argument; therefore @ must not be referenced. In addition, for all callback resources of the new widget class, the Destroy method calls XtRemoveAllCallbacks to free the storage allocated for the callback lists.

11.2.9 The Realize Method

If an active value with the name Realize exists for the model object of the class, its Set script is called in the class Realize method, with the following arguments:

@0 = value_mask parameter of the Realize method

@1 = attributes parameter of the Realize method

If you specify a Realize method, you must call a function which creates the widget's window, otherwise the X Toolkit would get confused and XFaceMaker could eventually crash. You can either call XtCreateWindow directly, or call the superclass's Realize method using the convenience function SuperclassRealize(w, value_mask, attributes)

```
SuperclassRealize(self, @, @1);
```

11.2.10 The SetValuesAlmost Method

If an active value with the name SetValuesAlmost exists for the model object of the class, its Set script is called in the SetValuesAlmost method, with the following arguments:

@0 = old parameter of the SetValuesAlmost method

@1 = request parameter of the SetValuesAlmost method

@2 = reply parameter of the SetValuesAlmost method

11.2.11 The AcceptFocus Method

If an active value with the name AcceptFocus exists for the model object of the class, its Set script is called in the class AcceptFocus method, with the following arguments:

@0 = time parameter of the AcceptFocus method

In addition, the AcceptFocus method of the class will return the contents of @1 which should be set by the Set script of the active value (@1 is initialized to False).

11.2.12 The QueryGeometry Method

If an active value with the name QueryGeometry exists for the model object of the class, its Set script is called in the class QueryGeometry method, with the following arguments:

@0 = request parameter of the QueryGeometry method

@1 = geometry_return parameter of the QueryGeometry method

In addition, the QueryGeometry method of the class will return the contents of @2 which should be set by the Set script of the active value (@2 is initialized to 0).

11.2.13 The DisplayAccelerators Method

If an active value with the name DisplayAccelerators exists for the model object of the class, its Set script is called in the DisplayAccelerators method, with the following arguments:

$ = string parameter of the DisplayAccelerators method

11.2.14 The GeometryManager Method

If an active value with the name GeometryManager exists for the model object of the class, its Set script is called in the GeometryManager method, with the following arguments:

@0 = w parameter of the GeometryManager method (i.e. the child widget for which a geometry request is made)

@1 = request parameter of the GeometryManager method

@2 = geometry_return parameter of the GeometryManager method

In addition, the GeometryManager method of the class will return the contents of @3 which should be set by the Set script of the active value (@3 is initialized to 0). Note that in the Set script of the GeometryManager active value, the self widget is the *parent* of the child for which a geometry request is made. The child widget, which is passed by the X Toolkit as the w argument to the GeometryManager method, can be accessed in @. The GeometryManager method can be set only for a sub–class of a Composite. If the model object for the class is not a sub–class of a Composite, a warning message is printed and the active value is ignored.

11.2.15 The ChangeManaged Method

If an active value with the name ChangeManaged exists for the model object of the class, its Set script is called in the class ChangeManaged method, with a $ NULL argument; therefore @ must not be referenced. The ChangeManaged method can be set only for a sub–class of a Composite. If the model object for the class is not of sub–class of a Composite, a warning message is printed and the active value is ignored.

11.2.16 The InsertChild Method

If an active value with the name InsertChild exists for the model object of the class, its Set script is called in the class InsertChild method. The $ argument passed to the script is the child widget w which must be inserted. The self widget is the parent widget which is an instance of the new class. The InsertChild method can be set only for a sub–class of a Composite. If the model object for the class is not a sub–class of a Composite, a warning message is printed and the active value is ignored. If you specify an InsertChild method, you must call a function which inserts the new widget in its parent's children array, otherwise the widget will not be managed by its parent. To do this, you can call the superclass's InsertChild method using the convenience function SuperclassInsertChild(parent, w)

```
SuperclassInsertChild(self, $);
```

11.2.17 The DeleteChild Method

If an active value with the name DeleteChild exists for the model object of the class, its Set script is called in the DeleteChild method. The $ argument passed to the script is the child widget w which must be deleted. self is the parent widget which is an instance of the new class. Note that the DeleteChild method can be set only for a sub–class of a Composite. If the model object for the class is not a sub–class of a Composite, a warning message is printed and the active value is ignored. If you specify the DeleteChild method, you must call a function which removes the widget from its parent's children array, otherwise you widget will still be managed by its parent and this could cause a crash. To do this, you can call the superclass's DeleteChild method using the convenience function SuperclassDeleteChild(parent, w)

```
SuperclassDeleteChild(self, $);
```

11.3 Implementation

11.3.1 The Class Record

The class record for the new class is an aggregate structure composed of the class parts of all the superclasses (including the class part of the model object class), plus the class part of the new widget class, which is empty.

All fields of the class record are initialized to the appropriate inheritance value, except the following fields of the Core class structure, which have values computed from the model object: **superclass**, **class_**name, **widget_size**, **class_initialize**, **initialize**, **actions, num_actions resources**, **num_resources**, **destroy**, **resize**, **expose**, **set_values get_**values_hook, **tm_**table.

11.3.2 The Widget Instance Record

The widget record for the new class is an aggregate of the widget parts of all the superclasses, plus the widget part of the new class. The new widget part contains an array containing the widget IDs of the widget's children, and locations where the values of the new resources, corresponding to the active values of the model object, will be stored.

CHAPTER 12: Functions

This chapter lists the functions available in XFaceMaker, alphabetically according to the function's C name, or its FACE name if there is no C equivalent. There are two tables at the beginning of the chapter, one listing functions that can only be called from a C program, the other listing those that can be called from either a C program or from a FACE script.

For each function, the type of the return value and of the arguments is given by the function prototype. Any function that can be called from a FACE script is shown with the FACE return type; e.g., Int, Pointer, etc.

Functions that can *only* be called from C programs are shown with their C return type; e.g., int, void, char, etc. The C prototype of each function is declared in the Fm.h include file.

All Motif functions can be called from FACE scripts, provided their arguments can be represented with FACE types. See your Motif Reference Manual for a description of the Motif functions. A complete list of the Motif functions is found in the Appendix.

The Xt functions that can be called from FACE scripts are listed in Table 12.2 and in the Appendix. Refer to your Xt Intrinsics Reference Manual for a description of the Xt functions.

12.1 The Application-Only Functions Table

The table on the next page, lists the functions that can *only* be called by the application and cannot be called in a FACE script, together with the libraries in which they can be found. An x indicates that the library contains the function.

C name	is the name of the function
Fm	the libFm.a library is the full standard library used by an application whose interface is in .fm format (interpreted mode interface).
Fm_c	the libFm_c.a library is the full library used by an application whose interface is in .c format (compiled mode interface).
Fm_e	the libFm_e.a library is a superset of the Fm library. It includes some additional functions which can be used to create an extended version of XFaceMaker.

As shown in the table, only a few application-only functions are included in the Fm_c library.

Table 12.1: Application-only Functions

C name	Fm	Fm_c	Fm_e
FmAddBootFile	x		x
FmAdd<ClassName>WidgetClass			
FmAddEnumeratedType	x		x
FmAddRepresentation	x		x
FmAddResourceType	x		x
FmAddWidgetClass	x		x
FmAppInitialize	x	x	x
FmAppInitializeC		x	
FmAppInitializeI	x		x
FmAttachAv	x	x	x
FmAttachFunction	x		x
FmAttachValue	x	x	x
FmCatClose	x	x	x
FmCatFind	x	x	x
FmCatGetS	x	x	x
FmCatOpen	x	x	x
FmChangeValueAddress	x	x	x
FmCreateManagedObject	x		x
FmCreateObject	x		x
FmDeleteVariable	x		x
FmEditLoop	x		x
FmFetchValue	x	x	x
FmFetchValueAdress	x	x	x
FmGetCatD	x	x	x
FmGetColorPixmap	x	x	x
FmGetFunctionsVector	x		x
FmInitialize	x	x	x
FmInitializeC		x	
FmInitializeI	x		x
FmLoad	x		x
FmLoadCreate	x		x

Table 12.1: Application-only Functions

C name	Fm	Fm_c	Fm_e
FmLoadCreateGroup	x		x
FmLoadCreateManaged	x		x
FmLoadCreateManagedGroup	x		x
FmLoadGroup	x		x
FmLoadWidgetClass	x		x
FmLoop	x	x	x
FmNewPredefinedVariable	x		x
FmNewType	x		x
FmNewTypeDef	x		x
FmReadValue	x	x	x
FmRegisterStructures	x		x
FmSetCloseHandler	x	x	x
FmSetFunctionPrototype	x		x
FmSetSuperclassInfos	x		x
FmSetToplevel	x	x	x
FmSetToplevelC		x	
FmSetToplevelI	x		x
FmWriteValue	x	x	x

12.2 The FACE Functions Table

The FACE Functions table lists the functions that are intended to be called from FACE scripts, although they can also be called directly from the application in their C name form, which can be a function name or an operator. Each entry includes the following information (from left to right):

FACE name is the name used in FACE scripts.

C name is the name that is generated in a C interface file, or that you can call directly from your application.

Class is the module to which the function belongs.

libraries are libraries which contain the function indicated by an x The libraries are the following:

Xm/Xt the standard (i.e., non-NSL) libraries, including the C library and the OSF/Motif libXm.a and libXt.a. Only those functions for which type checking is done are listed here. Refer to Appendix A for a full list of Xt or Xm functions that can be called from FACE scripts.

Fm the libFm.a library is the full library used by an application whose interface is in .fm format (interpreted mode interface).

Fm_c the libFm_c.a library is the full library used by an application whose interface is in .c format (compiled mode interface).

Fm_e the libFm_e.a library is a superset of the Fm library. It includes some additional functions which can be used to create new widget classes, and to call the editor from the application.

FACE name	C name	Class	Xm/Xt	Fm	Fm_c	Fm_e
AddGrab	XtAddGrab	Xt	x			
Beep	FmBeep	FACE		x	x	x
CallActionProc	XtCallActionProc	Xt	x			
CallCallbacks	XtCallCallbacks	Xt	x			
CallCallbacksTimeOut	FmCallCallbacksTimeOut	FACE		x	x	x
CallEditor	FmCallEditor	FACE				x
ClearArgList	FmClearArgList	FACE		x	x	x
ConfigureWidget	XtConfigureWidget	Xt	x			
ConvertString	FaceConvertString	FACE		x	x	x
CreateXmString	FmCreateXmString	FACE		x	x	x
DestroyWidget	XtDestroyWidget	Xt	x			
DisableTraversal	FmDisableTraversal	FACE		x	x	x
EnableTraversal	FmEnableTraversal	FACE		x	x	x
Exit	exit	C	x			
FaceInitTimer	FaceInitTimer	FACE		x		x
FaceStopTimer	FaceStopTimer	FACE		x		x
FmAddArg	FmAddArg	FACE		x	x	x
FmBeep	FmBeep	FACE		x	x	x
FmCallEditor	CallEditor	FACE				x
FmCallValue	FmCallValue	FACE		x	x	x
FmClearArgList	FmClearArgList	FACE		x	x	x
FmCreateDragIcon	FmCreateDragIcon	FACE		x	x	x
FmCreateRectangle	FmCreateRectangle	FACE		x	x	x
FmDeleteObject	FmDeleteObject	FACE				
FmDoEvent	FmDoEvent	FACE		x	x	x
FmDragStart	FmDragStart	FACE		x	x	x
FmDropSiteRegister	FmDropSiteRegister	FACE		x	x	x
FmDropSiteRetrieve	FmDropSiteRetrieve	FACE		x	x	x
FmDropSiteUpdate	FmDropSiteUpdate	FACE		x	x	x
FmDropTransferAdd	FmDropTransferAdd	FACE		x	x	x
FmDropTransferStart	FmDropTransferStart	FACE		x	x	x
FmEqualString	FmEqualString	FACE		x	x	x
FmFreeArgList	FmFreeArgList	FACE		x	x	x
FmGetPixmap	FmGetPixmap	FACE		x	x	x
FmGetStringfromTable	FmGetStringFromTable	FACE		x	x	x

Table 12.2: FACE Functions

FACE name	C name	Class	Xm/Xt	Fm	Fm_c	Fm_e
FmGetValue	FmGetValue	FACE		x	x	x
FmGetXmDisplay	FmGetXmDisplay	FACE		x	x	x
FmGetXmScreen	FmGetXmScreen	FACE		x	x	x
FmHideWidget	FmHideWidget	FACE		x	x	x
FmListAllowKeySelection	FmListAllowKeySelection	FACE		x	x	x
FmListGetItems	FmListGetItems	FACE		x	x	x
FmListGetSelectedItems	FmListGetSelectedItems	FACE		x	x	x
FmNewArgList	FmNewArgList	FACE		x	x	x
FmRegisterSimpleDropSite	FmRegisterSimpleDropSite	FACE		x	x	x
FmSendClick	FmSendClick	FACE		x	x	x
FmSetTallest	FmSetTallest	FACE		x	x	x
FmSetValue	FmSetValue	FACE		x	x	x
FmSetWidest	FmSetWidest	FACE		x	x	x
FmStartCursorDrag	FmStartCursorDrag	FACE		x	x	x
FmStartSimpleDrag	FmStartSimpleDrag	FACE		x	x	x
FmWidest	FmWidest	FACE		x	x	x
FreeArgList	FmFreeArgList	FACE		x	x	x
GetActiveValue	FmGetActiveValue	FACE		x	x	x
GetActiveValueAddr	FmGetActiveValueAddr	FACE		x	x	x
GetActiveValueTimeOut	FmGetActiveValueTimeOut	FACE		x	x	x
GetString	FmGetString	FACE		x	x	x
GetVersionString	FmGetVersionString	FACE		x	x	x
Hide	FmHideWidget	FACE		x	x	x
Internationalize	FmInternationalize	FACE		x	x	x
IsManaged	XtIsManaged	Xt	x			
IsSensitive	XtIsSensitive	Xt	x			
ListGetNthItem	FmListGetNthItem	FACE		x	x	x
ListGetNthSelectedItem	FmListGetNthSelectedItem	FACE		x	x	x
ListSetItems	FmListSetItems	FACE		x	x	x
LowerWidget	FmLowerWidget	FACE		x	x	x
ManageChild	XtManageChild	Xt	x			
MapWidget	XtMapWidget	Xt	x			
MoveWidget	XtMoveWidget	Xt	x			
NewArgList	FmNewArgList	FACE		x	x	x
NewString	XtNewString	Xt	x			
Popdown	XtPopdown	Xt	x			
Popup	XtPopup	Xt	x			

Table 12.2: FACE Functions

FACE name	C name	Class	Xm/Xt	Fm	Fm_c	Fm_e
PopupAndWait	FmPopupAndWait	FACE		x	x	x
Quit	exit	C	x			
RaiseWidget	FmRaiseWidget	FACE		x	x	x
RemoveGrab	XtRemoveGrab	Xt	x			
ResizeWidget	XtResizeWidget	Xt	x			
Return	FmReturn	FACE		x	x	x
SendClick	FmSendClick	FACE		x	x	x
SendMessage	FmSendMessage	FACE		x	x	x
SetActiveValue	FmSetActiveValue	FACE		x	x	x
SetActiveValueTimeOut	FmSetActiveValueTimeOut	FACE		x	x	x
SetKeyboardFocus	XtSetKeyboardFocus	Xt	x			
SetSensitive	XtSetSensitive	Xt	x			
SetValues	FaceSetValues	FACE		x	x	x
Show	FmShowWidget	FACE		x	x	x
ShowPopup	FmShowPopup	FACE		x	x	x
StartSimpleDrag	FmStartSimpleDrag	FACE		x	x	x
StopTimeOut	FmStopTimeOut	FACE		x	x	x
SuperclassDeleteChild	SuperclassDeleteChild	FACE		x		x
SuperclassExpose	SuperclassExpose	FACE		x		x
SuperclassInsertChild	SuperclassInsertChild	FACE		x		x
SuperclassRealize	SuperclassRealize	FACE		x		x
SuperclassResize	SuperclassResize	FACE		x		x
UnmanageChild	XtUnmanageChild	Xt	x			
UnmapWidget	XtUnmapWidget	Xt	x			
WaitForReturn	FmWaitForReturn	FACE		x	x	x
WarpPointer	FmWarpPointer	FACE		x	x	x
WidgetName	FmWidgetName	FACE		x	x	x
atoi	atoi	C	x			
breakpoint		FACE				x
callback	XtCallCallbacks	Xt	x			
cbtoi		FACE				
cftoi		FACE				
citof		FACE				
clear		FACE				
cstoi		FACE				
data		FACE		x	x	x
destroy		FACE		x	x	x

Table 12.2: FACE Functions

FACE name	C name	Class	Xm/Xt	Fm	Fm_c	Fm_e
dtoi	FmDtoi	FACE		x	x	x
equal	FmEqualString	FACE		x	x	x
eval_string	FaceEvalString	FACE		x		x
fclose	fclose	C	x			
fgets	fgets	C	x			
first		FACE		x	x	x
fopen	fopen	C	x			
fputs	fputs	C	x			
free	free	C	x			
fscanf	fscanf	C	x			
ftoi	FmFtoi	FACE		x	x	x
hide	FmHideWidget	FACE		x	x	x
init_timer	FaceInitTimer	FACE		x		x
itoa	FmItoa	FACE		x	x	x
itod	FmItod	FACE		x	x	x
itof	FmItof	FACE		x	x	x
malloc	malloc	C	x			
new_array		FACE		x	x	x
new_c_array		FACE		x	x	x
not_equal	FmNotEqualString	FACE		x	x	x
new_struct		FACE		x	x	x
new_table		FACE		x	x	x
next		FACE		x	x	x
pclose	pclose	C	x			
popen	popen	C	x			
printf	printf	C	x			
random		FACE		x		x
return		C	x			
send	FmSendClick	FACE		x	x	x
send_click	FmSendClick	FACE		x	x	x
show	FmShowWidget	FACE		x	x	x
size		FACE		x	x	x
sprintf	sprintf	C	x			
sscanf	sscanf	C	x			
stop_timer	FaceStopTimer	FACE		x		x
strcat	strcat	C	x			
strcmp	strcmp	C	x			

Table 12.2: FACE Functions

FACE name	C name	Class	Xm/Xt	Fm	Fm_c	Fm_e
strcpy	strcpy	C	x			
strlen	strlen	C	x			
strncmp	strncmp	C	x			
wait	FmWait	FACE		x	x	x

Table 12.2: FACE Functions

12.3 Function Reference Pages

The format for the functions in this chapter is the following:

C NAME **FACE NAME**

The C name of the function. **The FACE name of the function**

SYNOPSIS

The calling syntax, the arguments, and the return type.

DESCRIPTION

A brief description of the function.

SCOPE

States whether the function can be called from a FACE script only, from a FACE script and the application, or from the application only.

EXAMPLE

An example of its use.

SEE ALSO

A list of related functions.

C NAME FACE NAME

FaceConvertString **ConvertString**

SYNOPSIS

```
Any FaceConvertString (
        Widget      w,
        String      from,
        String      to_type)
```

DESCRIPTION

Explicitly convert the string *from* to the to_type resource type. This function calls XtConvert. For example:

```
pixmap=ConvertString(self,"50_foreground", "Pixmap");
```

converts the string "50_foreground" into the corresponding Pixmap. The return type of the function is “Any”, which means that type checking will not be done on the result unless the appropriate type caster is used:

```
pix=Pixmap(ConvertString(..., ..., ...));
```

SCOPE

FACE scripts application

EXAMPLE

Redefine the translation for <KeyPress> osfLeft; some managers (e.g. row-column) redefine the translation table for some keys, such as osfLeft; in that case, you can’t use the translation resource to redefine the translation table. You can redefine it with a script such a this one in the widget’s create callback.

```
t=ConvertString(self.XmPushButton0,
        ’#override <KeyPress>osfLeft:eval("Beep();")’
        "TranslationTable”);
self.XmPushButton0:translations = t;
```

SEE ALSO

XtConvert, XtConvertAndStore

C NAME FACE NAME

FaceEvalString **eval_string**

SYNOPSIS

```
int FaceEvalString(
        Widget      w,
        String      *str,
        Pointer     param)
```

DESCRIPTION

Interpret the FACE script whose text is in *str*, a character string, e.g. "i = 1;" and return -1 if the script was incorrect. Execution of an incorrect script causes an error message to be sent to stderr. This function is not available in the library libFm_c.a.

widget is the reference widget, i.e., the widget which determines the content of self and of parent. It need not be otherwise related to the script being evaluated unless it is directly referenced by the script.

param is any parameter. It has been declared here as an *int* but could be any legal FACE type. In a FACE script, this parameter is referenced by the read-only variable $.

SCOPE

FACE scripts application

EXAMPLE

```
eval_string(self,"st=WidgetName(parent.PB_2);
                                        printf('%s\n',st);",Pointer(2));
```

SEE ALSO

C NAME — FACE NAME

FaceInitTimer — **init_timer**

SYNOPSIS

```
Timer init_timer (
        Widget      w,
        String      script,
        None        call_data,
        Int         delay,
        Boolean     auto-repeat)
```

DESCRIPTION

Initialize a timer so that the script *script* will be called automatically in *delay* milliseconds. *script* is a string containing a FACE script. After *delay* milliseconds, the script will be parsed and executed, using *w* as a reference Widget, and *call_data* as the $ argument. If *auto-repeat* is True, then the timer is re-initialized automatically after the script is executed, with the same delay. The return value is an identifier which can be passed to stop_timer to cancel the timer before it has expired. This function is not available in the library libFm_c.a.

SCOPE

FACE scripts application

EXAMPLE

In a pushbutton, set the label to 0 and increment it every half second as long as the button is armed.

```
armCallback = global t; global i;i=0;
        self:labelString=itoa(i);
t = init_timer(self, "global i; i=i+1;
        self:labelString=itoa(i);",0,500,True); disarmCallback = global t;
stop_timer(t);
```

SEE ALSO

FaceStopTimer

XtAddTimeOut

FmCallCallbacksTimeOut

FmStopTimeOut

C NAME | FACE NAME

FaceSetValues **SetValues**

SYNOPSIS

```
Int FaceSetValues(
        Widget      w,
        Int         num_resources,
        String      resource1,
        None        value1,
        String      resource2,
        None        value2,
        ...)
```

DESCRIPTION

Set multiple resource values in one call to XtSetValues.

num_resources specifies the number of resources to be set. A maximum of 5 resources can be set simultaneously.

resource1 is the name of the first resource.

value1 is the value to which it will be set.

The values must be given in their internal form, not in their String form. The internal form of a resource can be obtained by using ConvertString.

SCOPE

FACE scripts   application

EXAMPLE

Set the resources for the scale widget in a float resource editing box:

```
SetValues(parent.scale, 3,
"minimum", cftoi(min * 100.0),
"maximum", cftoi(max * 100.0),
"value", cftoi(v * 100.0));
```

SEE ALSO

ConvertString

XtSetValue

C NAME **FACE NAME**

FaceStopTimer **stop_timer**

SYNOPSIS

```
void FaceStopTimer(
        Timer         timer)
```

DESCRIPTION

Stop a timer started by FaceInitTimer. FaceStopTimer is not available in the library libFm_c.a.

timer is the timer returned from the call to FaceInitTimer for the timer you want to stop.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FaceInitTimer

C NAME FACE NAME

FmAddArg **FmAddArg**

SYNOPSIS

```
None FmAddArg (
        FmArgList   list,
        String      name,
        None        value)
```

DESCRIPTION

Add an argument to an argument list.

list is a list of arguments created with FmNewArgList.

name is a resource name.

value is the value of the resource with its internal representation.

SCOPE

FACE scripts application

EXAMPLE

Taken from the DragDrop example in the distribution:

```
FmArgList arglist = FmNewArgList();
FmAddArg(arglist,"pixmap",
        FmGetPixmap(self,"letters.bm"));
FmAddArg(arglist,"mask",ConvertString
                (self,"letters_m.bm","Bitmap"));
Widget source_icon = FmCreateDragIcon(self,
                                "source",arglist);
FmClearArgList(arglist);
```

SEE ALSO

FmNewArgList

FmClearArgList

FmFreeArgList

C NAME   FACE NAME

FmAddBootFile   **none**

SYNOPSIS

```
None FmAddBootFile (
          char          *filename)
```

DESCRIPTION

Specify additional .fm files that must be loaded after the Fm.fm file as part of the XFaceMaker user interface.FmAddBootFile is not available in the library libFm_c.a.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME | FACE NAME

FmAddEnumeratedType | **none**

SYNOPSIS

```
Boolean FmAddEnumeratedType (
        char        *type,
        char        *values,
        int         offset,
        char        *prefix)
```

DESCRIPTION

When adding a new widget class to XFaceMaker, FmAddEnumeratedType can be used to add a new enumerated resource type instead of FmAdd-ResourceType. This allows a prefix to be defined for the enumeration values in place of the default Xm. The function returns 1 on success, 0 on failure.

type specifies the representation name of the new resource type, i.e. XtRNew-Type, XmRNewType, or XNewRNewType.

values is a character string that defines the string values of the enumeration. each value must end with the character \n. For example:

"choice1\nchoice2\nchoice3\n"

offset specifies the integer value of an enumerated resource value. The values following must be contiguous. Therefore, if *offset* is 3, then "choice1" is 3, "choice2" is 4 and "choice3" is 5.

prefix is a character string that defines the prefix for the enumeration values, i.e., "Xnsl" giving "XnslCHOICE1" which has integer value 0.

SCOPE

Application only

EXAMPLE

```
FmAddEnumeratedType(XnslRNewType2,
        "choice1\nchoice2\nchoice3\n",10,"Xnsl");
```

SEE ALSO

FmAddResourceType

C NAME / FACE NAME

FmAddRepresentation **none**

SYNOPSIS

```
int         FmAddRepresentation (
            const       char *type,
            const       char *repname,
            const       char *ctype,
            int         size,
            int         aligned,
            int         indirect)
```

DESCRIPTION

This function informs XFaceMaker on how to manipulate resources for a new resource type, and how to generate C code when the resource is set for a widget or used in a FACE script. The arguments are:

type The type name (e.g. “MyType”).

repname The representation name symbol in C, i.e. the name of the C symbol containing the type name in a C program (e.g. “XnslRMyType”). The representation name is used to generate portable C code with references to the representation symbol, which may have different values on different platforms (e.g. the representation name XmFontList is defined as “FontList” on most platforms but as “XmFontList” on DEC platforms).

ctype The C type declarator for a C variable usable as the value of the resource (e.g. “struct my_type *”).

size The size of the resource value (on the current platform) (e.g. sizeof(struct my_type *)).

aligned Value is 0 or 1. If 1, means that the resource value is guaranteed to be always the same size as an XtArgVal (on all platforms). This can be assumed as true for all pointers on all known platforms but is not guaranteed to be always the case.

indirect reserved -- pass 0.

The function returns *0*, or *1* if it has increased the size of the new types table. The number of new types that can be defined is not limited, the table is reallocated if the current table is too small. The increment for the size of the table is 20.

SCOPE

Application only

EXAMPLE

```
FmAddRepresentation(XtRXrtData, “XtRXrtData”, “XrtData*”,
                    sizeof(XrtData*), 1, 0);
```

SEE ALSO

C NAME

FmAddResourceType

FACE NAME

none

SYNOPSIS

```
Boolean FmAddResourceType (
        char        *type,
        char        get_value,
        char        avtype,
        char        *avname,
        char        *boxname,
        char        *default_value,
        char        *enumeration,
        int         offset)
```

DESCRIPTION

FmAddResourceType can be used to add a new resource type which is required by a widget class added to XFaceMaker with FmAddWidgetClass.

The function must be called after FmAddWidgetClass and before FmCallEditor. You cannot add a resource type that is already declared in XFaceMaker: see Table 12.3, "Predefined Resource Types".

Note that FmAddEnumeratedType is a special version of FmAddResourceType, for enumerated types only.

type specifies the representation name of the new resource type, i.e., XtRNewType, XmRNewType, or XNewRNewType.

get_value defines how the current resource value is to be retrieved:

RT_WIDGET the current value of the resource is retrieved from the widget using XtGetValues.

[**RT_OBJECT** the current value is fetched from the resources of the XFaceMaker object, and avtype must be **RT_STRING**.

RT_WIDGET is valid only for integer or string resources; for other resources, RT_OBJECT must be used.

avtype specifies how XFaceMaker will allocate the storage that will be attached to the active value *avname*:

RT_INTEGER The active value associated to the resource type will be attached to the address of a memory location large enough to hold an integer or a pointer.

RT_STRING The active value associated to the resource type will be attached to the address of an 8 KBytes buffer holding the value of the resource as a string.

Thus, for integer or other immediate resources like floats, the value will be the contents of the Active Value variable @, whereas for string resources, the string value is $.

avname specifies the name of the active value attached to the resource type. If *avname* is NULL, then the *type* parameter is used.

boxname specifies the name of a Shell widget in the XFaceMaker interface to pop up when a resource of this type is to be edited. This can be either an existing box, or a new one defined in a customized Fm.fm file. If *boxname* is NULL, the *type* box is used.

default_value specifies a default value to display in the editing box if no value has been defined yet for the resource. The default value can be NULL.

enumeration specifies the values will be displayed in the list of the Enumeration box if the *boxname* is “Enumeration”. Each value must end with the character \n . For example: “choice1\nchoice2\nchoice3\n”

If boxname is not “Enumeration” this argument must be NULL. Alternatively you can use the function FmAddEnumeratedType.

offset specifies the integer value of an enumerated resource type. The values following must be contiguous. Therefore if offset is 3, then “choice1” is 3, “choice2” is 4 and “choice3” is 5.

FmAddResourceType returns 1 on success, 0 on failure. For example:

```
FmAddResourceType(“NewType1”
        RT_OBJECT,
        RT_STRING,
        0, 0, 0, 0, 0))
```

This call declares the new resource type “NewType1”. When a resource of this type is selected in the XFaceMaker resource list, an active value called “NewType1” will be set, and the shell called “NewType1” will pop up.

Table 12.3, “Predefined Resource Types”lists the resource types already defined in XFaceMaker, with their parameters. You can use the existing editing boxes for your new resource types, but in that case the parameters *get_val*, *avtype* and *avname* must match those of the existing types.

Table 12.3: Predefined Resource Types

Type	G	A	Avname	Boxname	Enumeration (1st)	o
AcceleratorTable	O	S	Translation-Table	Translations		0
Alignment	W	I	Enumeration	Enumeration	alignment_beginning	0
AnimationStyle	W	I	Enumeration	Enumeration	drag_under_none	0
ArrowDirection	W	I	Enumeration	Enumeration	arrow_up	0
Attachment	W	I	Enumeration	Enumeration	attach_none	0
AudibleWarning	W	I	Enumeration	Enumeration	none	0
Bitmap	O	S	Image	Images		0
BlendModel	W	I	Enumeration	Enumeration	blend_all	0
Bool	W	I	Boolean	Boolean		0
Boolean	W	I	Boolean	Boolean		0
BooleanDimension	W	I	PositiveInt	Integer		0
Callback	O	S	0	TextEdit		0
Cardinal	W	I	PositiveInt	Integer		0
ChildHorizontalAlignment	W	I	Enumeration	Enumeration	alignment_beginning	0
ChildPlacement	W	I	Enumeration	Enumeration	place_top	0
ChildType	W	I	Enumeration	Enumeration	frame_generic_child	0
ChildVerticalAlignment	W	I	Enumeration	Enumeration	alignment_baseline_top	0
CommandWindowLocation	W	I	Enumeration	Enumeration	command_above_workspace	0
Cursor	O	S	String	String		0
DefaultButtonType	W	I	Enumeration	Enumeration	dialog_none	0
DeleteResponse	W	I	Enumeration	Enumeration	destroy	0
DialogType	W	I	Enumeration	Enumeration	dialog_template	0
DialogStyle	W	I	Enumeration	Enumeration	dialog_modeless	0
Dimension	W	I	PositiveInt	Integer		0
DragInitiatorProtocolStyle	W	I	Enumeration	Enumeration	drag_none	0
DragReceiverProtocolStyle	W	I	Enumeration	Enumeration	drag_none	0
DropSiteActivity	W	I	Enumeration	Enumeration	drop_site_active	0
DropSiteType	W	I	Enumeration	Enumeration	drop_site_simple	0
EditMode	W	I	Enumeration	Enumeration	multi_line_edit	0
ExtensionType	W	I	Enumeration	Enumeration	cache_extension	0
Float	W	I	0	0		0

G = get_value; A = avtype; o = offset; O=RT_OBJECT; S=RT_STRING; W = RT_WIDGET; I = RT_INTEGER;

Table 12.3: Predefined Resource Types

Type	G	A	Avname	Boxname	Enumeration (1st)	o
FileTypeMask	W	I	Enumeration	Enumeration	file_directory	1
Font	O	S	FontList	Fonts		0
FontList	O	S	FontList	Fonts		0
FontStruct	O	S	FontList	Fonts		0
GadgetPixmap	O	S	Image	Images		0
HorizontalDimension	W	I	PositiveInt	Integer		0
HorizontalPosition	W	I	Int	Integer		0
HorizontalInt	W	I	Int	Integer		0
IconAttachment	W	I	Enumeration	Enumeration	attach_north_west	0
Image	O	S	0	Images		0
IndicatorType	W	I	Enumeration	Enumeration	n_of_many	1
InitialState	W	I	Enumeration	Enumeration	NormalState	1
Int	W	I	0	Integer		0
KeyboardFocusPolicy	W	I	Enumeration	Enumeration	explicit	0
LabelType	W	I	Enumeration	Enumeration	pixmap	1
ListSizePolicy	W	I	Enumeration	Enumeration	variable	0
ManBottomShadowPixmap	O	S	Image	Images		0
ManForegroundPixmap	O	S	Image	Images		0
ManHighlightPixmap	O	S	Image	Images		0
ManTopShadowPixmap	O	S	Image	Images		0
MenuWidget	O	S	Widget	Widget		0
MultiClick	W	I	Enumeration	Enumeration	multiclick_discard	0
NavigationType	W	I	Enumeration	Enumeration	none	0
Orientation	W	I	Enumeration	Enumeration	vertical	1
Packing	W	I	Enumeration	Enumeration	pack_tight	1
Pixel	O	S	0	Colors		0
Pixmap	O	S	Image	Images		0
Position	W	I	Int	Integer		0
PrimBottomShadowPixmap	O	S	Image	Images		0
PrimForegroundPixmap	O	S	Image	Images		0
PrimHighlightPixmap	O	S	Image	Images		0
PrimTopShadowPixmap	O	S	Image	Images		0
ProcessingDirection	W	I	Enumeration	Enumeration	max_on_top	0
ResizePolicy	W	I	Enumeration	Enumeration	resize_none	0

G = get_value; A = avtype; o = offset; O=RT_OBJECT; S=RT_STRING; W = RT_WIDGET; I = RT_INTEGER;

Table 12.3: Predefined Resource Types

Type	G	A	Avname	Boxname	Enumeration (1st)	o
RowColumnType	W	I	Enumeration	Enumeration	work_area	0
ScrollBarDisplayPolicy	W	I	Enumeration	Enumeration	static	0
ScrollBarPlacement	W	I	Enumeration	Enumeration	bottom_right	0
ScrollingPolicy	W	I	Enumeration	Enumeration	automatic	0
SelectionPolicy	W	I	Enumeration	Enumeration	single_select	0
SelectionType	W	I	Enumeration	Enumeration	dialog_work_area	0
SeparatorType	W	I	Enumeration	Enumeration	no_line	0
ShadowType	W	I	Enumeration	Enumeration	shadow_etched_in	5
ShellHorizDim	W	I	PositiveInt	Integer		0
ShellHorizPos	W	I	Int	Integer		0
ShellUnitType	W	I	Enumeration	Enumeration	pixels	0
ShellVertDim	W	I	PositiveInt	Integer		0
ShellVertPos	W	I	Int	Integer		0
Short	W	I	Int	Integer		0
SizePolicy	W	I	Enumeration	Enumeration	change_all	0
String	W	S	0	0		0
StringDirection	W	I	Enumeration	Enumeration	string_direction_l_to_r	0
StringTable	O	S	_FmTable	Table		
TearOffModel	W	I	Enumeration	Enumeration	tear_off_enabled	0
TextPosition	W	I	Int	Integer		0
TopItemPosition	W	I	PositiveInt	Integer		0
TranslationTable	O	S	0	Translations		0
UnitType	W	I	Enumeration	Enumeration	pixels	0
UnpostBehavior	W	I	Enumeration	Enumeration	unpost	0
VerticalAlignment	W	I	Enumeration	Enumeration	alignment_baseline_top	0
VerticalDimension	W	I	PositiveInt	Integer		0
VerticalInt	W	I	Int	Integer		0
VerticalPosition	W	I	Int	Integer		0
VisualPolicy	W	I	Enumeration	Enumeration	variable	0
Widget	O	S	Widget	Widget		0
Window	O	S	Widget	Widget		0
XmBackgroundPixmap	O	S	Image	Images		0
XmString	O	S	XtRString	XtRString		0
XmStringTable	O	S	_FmTable	Table		0

G = get_value; A = avtype; o = offset; O=RT_OBJECT; S=RT_STRING; W = RT_WIDGET; I = RT_INTEGER;

SCOPE

Application only

EXAMPLE

SEE ALSO

FmAddWidgetClass
FmAddBootFile

C NAME   FACE NAME

FmAddWidgetClass **none**

SYNOPSIS

```
Boolean FmAddWidgetClass (
WidgetClass                         *widget_class,
WidgetClass                         *gadget_class,
FmWidgetType                        type,
unsigned long                       flags,
char                                *default_resources,
char                                *include_file,
char                                *class_ref,
char                                *gadget_class_ref,
XtResourceList                      sub_resources,
Cardinal                            num_sub_resources,
char                                *icon_bitmap_file)
```

DESCRIPTION

Add a new widget class to the set of classes that already XFaceMaker already knows.The parameters to the function describe the characteristics of the new class. All these parameters can be 0, which specifies the default case, except widget_class which must be non-zero. FmAddWidgetClass returns 1 on success, 0 on failure.

Once a new Widget class is added, it can be used in XFaceMaker exactly in the same way as standard OSF/Motif classes. The only difference being that unless the function FmAddResourceType, or FmAddEnumeratedType, is used to define resource types for the new widget, resources that have a type which is not known by XFaceMaker will be edited with the String resource edit box.

widget_class is the address of the WidgetClass pointer, defined in the header file of the new class. For instance:

```
&myNewWidgetClass
```

gadget_class is the address of the GadgetClass pointer, if the new class supports gadgets (0 otherwise).

type specifies what category the new widget class belongs to (Primitive, Composite, etc...). The possible values are:

FM_PRIMITIVE for a Primitive Widget.

FM_COMPOSITE for a Composite Widget

FM_SHELL for a Shell Widget

FM_MENUCASCADE for a Menu Cascade Widget

FM_APPSHELL for an Application Shell Widget

unsigned long flags specifies special characteristics of the new widget class. The values are defined and explained below; however, the value **FM_NORMAL** (or 0) can be used most of the time. Possible values are:

FM_NORMAL normal case.

FM_NEED_REBUILD for Composite widgets which must be rebuilt when a child changed.

FM_ONE_EXPOSE for widgets which redraw on expose event hence clear all selection handles in one go.

FM_ONE_CHILD only one child widget is allowed.

FM_NO_CHILD no child widgets are allowed.

FM_INIT_CLASS the class must be initialized before the first instantiation; this flag is no longer necessary because XFaceMaker now always initializes a widget class before it first creates an instance of that class; the flag remains only for compatibility.

FM_KEY_SENS widget is sensitive to Key events. This means that XFaceMaker cannot intercept characters in this widget and interpret them as its own accelerator keys.

FM_DONT_FREE_VALUES specifies that XmString resource values returned by GetValues should not be freed by XFaceMaker when it lists current resource settings in the resource window. This is because some third-party widgets do not return a copy of an XmString resource the way Motif widgets do.

default_resources specifies the default resource values that will be used each time a widget instance is created. Resource assignments have the same format as in the Resource Editor window of XFaceMaker, and must be newline-separated. For example:

“width=10\nheight=100\n”

If there are no default resources, pass 0. If you have a widget with resources which cannot be set using strings, either rewrite the widget to use sensible default values or use xrdb to load a default resources file before launching XFaceMaker.

include_file is a string used to generate the C-code file containing instances of the new widget class, and specifies the header file that must be included in the generated C file. The whole line must be specified; e.g.:

“#include <MyWidget.h>”

char **class_ref* is also used to generate the C-code file, and is the C name of the WidgetClass pointer for the new class, for instance:

“myNewWidgetClass”

char **gadget_class_ref* is the C name of the GadgetClass pointer (if it exists), or the null pointer.

sub_resources specifies additional resources not listed in the resources field of the new WidgetClass record. Some widget classes (like XmText, for instance) use special resources that are accessed using the XtGetSubresources function. Such resources must be specified here in order to be usable in XFaceMaker, as an array of XtResource records where only the following fields have to be filled: resource_name, resource_class, resource_type, resource_size. If the new widget class does not use sub-resources, pass 0.

num_sub_resources specifies the number of sub-resources in the *sub_resources* array.

icon_bitmap_file specifies the bitmap file to be used for the new icon in the Widget Toolkit.Their size is typically 32x32. If NULL is passed, a default icon is used. If “xfm_no_icon” is passed, no icon is used.

FmAddWidgetClass returns 1 on success, 0 on failure. If successful, the new widget class can be used like other classes, and has the name contained in the WidgetClass record.

During the next call to the FmCallEditor function, a new icon will be created for the new widget class in the User defined panel of the Toolkit. FmAddWidgetClass can be used several times in the same program to add several new widget classes. XFaceMaker enforces no limit to the number of classes added.

SCOPE

Application only

SEE ALSO

FmAddResourceType

C NAME	**FACE NAME**
FmAppInitialize	**none**
FmAppInitializeC	**none**
FmAppInitializeI	**none**

SYNOPSIS

```
Widget      FmAppInitialize(
            XtAppContext          *app_context_ret,
            String                class,
            XrmOptionDescRec      ops[],
            Cardinal              num_options,
            int                   *argc,
            String                argv[],
            String                fallback_resources[],
            ArgList               args,
            Cardinal              num_args)
```

DESCRIPTION

Initialize all data structures required by XFaceMaker and call XtAppInitialize. The return value is the default ApplicationShell returned by XtAppInitialize. You must call FmAppInitialize (or FmInitialize) at the beginning of the application, not XtAppInitialize (or XtInitialize), unless you have generated the C code interface with the standalone option.

FmAppInitialize has the same arguments as XtAppInitialize and you should refer to the X documentation for a full explanation.

If your application does not need to use the application context returned, you can pass 0. An application which uses FmAppInitialize can use FmLoop for its main loop. The application context created by FmAppInitialize will be automatically used by the event loop.

The line:

```
top=FmAppInitialize
(0,"Name",NULL,0,&argc,argv,NULL,NULL,0);
```

is sufficient for an application called name which does not need to parse the command line to recognize its own options, to specify fallback resources, nor to pass an argument list. If you pass the class Xfm to FmAppInitialize, it will be replaced by XfmApplication. Otherwise, the class name that you pass will be the class name of the application you are building.

The FmAppInitialize function is also available in two versions: FmAppInitializeI is defined in the libFm_e and libFm libraries, while FmAppInitializeC is defined in the libFm_c library. The arguments and return values are the same as for FmAppInitialize.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmInitialize

C NAME FACE NAME

FmAttachAv **none**

SYNOPSIS

```
int FmAttachAv(
        char          *name,
        XtPointer     address)
```

DESCRIPTION

FmAttachAv associates the application variable whose address is specified to the active value called *name*. name must be an active value with global storage. If *name* exists already, it returns 1, otherwise 0.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmAttachValue

C NAME **FACE NAME**

FmAttachFunction **none**

SYNOPSIS

```
None FmAttachFunction(
            char        *name,
            int         (*func)(),
            char        *ret_typ,
            int         nb_args,
            char        *arg1_typ,
            char        *arg2_typ,
            ...)
```

DESCRIPTION

Attach an application function address to the interface function name, and provide information on the return value and parameter types so that the FACE interpreter can do type checking.

name is the name of the function as used in the FACE script.

func is the address of the function attached to it in the application.

ret_type is the type of the return value of the function or "None".

nb_args is the number of arguments to the function, with a maximum of twelve, or zero. If the *number_of_arguments* is given as -1, there is no argument checking when the function is called.

arg1_type is the type of first argument if present,

arg2_type is the type of second argument if present, up to *nb_args.*

Function arguments can be the basic FACE objects, other function calls, or arithmetic expressions. The type of these arguments can be any of the valid FACE types expressed as character strings, i.e., "String", "Widget", "Pointer", etc.

When an application function is called from a FACE script, the type of the arguments used in the call statement must agree with the typed specified when the function is attached to FACE. There is no automatic or implicit type conversion in function calls. It must have the same number of arguments as was specified or an error message will be given. You can, however, prevent argument type checking by using the special types "None" and "Any.

FmAttachFunction should be used when an interface is loaded by the application for use in interpreted mode. Functions must be attached before the interface fragment using them is loaded.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmLoadCreate

C NAME **FACE NAME**

FmAttachValue **none**

SYNOPSIS

```
int     FmAttachValue(
                Widget                          widget)
                char                            *name
                XtPointer                       address
```

DESCRIPTION

This function attaches an address to an active value for a given widget. If widget is NULL, this is equivalent to FmAttachAv and the function returns 1. Otherwise, if the active value has per-widget storage, the address is attached to the active value for the given widget only. If the active value has global storage, the address is attached globally as with FmAttachAv. In both cases, the return value is 2. If the active value does not exist, the function does nothing and returns 0.

widget is the identifier of the widget for which the active value is being attached, for an active value with "per-widget" storage. If the active value has global storage, pass NULL.

name is the name of the active value in the interface.

address is the address of the application variable associated to the active value if immediate storage is not used. If immediate storage is used, *address* is the contents of the active value. *address* will be passed as the $ read-only variable to the active value's set or get scripts.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmAttachAv

C NAME FACE NAME

FmBeep **Beep**

SYNOPSIS

None FmBeep()

DESCRIPTION

Produce a sound. Calls XBell.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

FmCallCallbacksTimeOut **CallCallbacksTimeOut**

SYNOPSIS

```
Pointer FmCallCallbacksTimeOut(
                Widget      w,
                String      callback,
                Int         interval)
```

DESCRIPTION

Initialize a timeout. *callback* is called in *interval* milliseconds by:
XtCallCallbacks(widget, callback, (caddr_t)0).
Returns a pointer to a timeout identifier TimeoutID. The timeout can be stopped with: StopTimeOut(Pointer TimeoutID).

SCOPE

FACE scripts application

EXAMPLE

Initiate a process in the create callback of the parent of a pushbutton; continue it in the pushbutton's activateCallback. Stop when i reaches 9. The process 'raises' label widgets successively.

```
xfmCreateCallback = global iter;
global buf;
iter=0;
buf = malloc(10);
sprintf(buf,"*Label0_%d",iter);
iter=iter+1;
 w = XtNameToWidget(self,buf);
RaiseWidget(w); CallCallbacksTimeOut(self.pb,"activateCallback",2000);
activateCallback = global iter;
global buf;
sprintf(buf,"*Label0_%d",iter);
iter=iter+1;
w = XtNameToWidget(root,buf);
RaiseWidget(w);
if (iter <=9)
CallCallbacksTimeOut(self,"activateCallback",2000);
else
iter=0;
```

SEE ALSO

FmStopTimeOut

FmGetActiveValueTimeOut

FmSetActiveValueTimeOut

C NAME **FACE NAME**

FmCallEditor **CallEditor**

SYNOPSIS

None FmCallEditor ()

DESCRIPTION

Call the XFaceMaker editor from an interpreted interface. If the application uses FmEditLoop as the main events handling function, the CallEditor function is also activated by typing
<Shift-Control-Escape>

while the pointer is in one of the windows of the interface. However,
<Shift-Control-Escape>

does not work in a window which was popped by PopupAndWait.

This function can also be called from an application as FmCallEditor, to produce a customized version of XFaceMaker. Any application or interface which calls this function must be linked with the library libFm_e. This library is protected and the resulting executable will expect the token server to be running.

SCOPE

FACE scripts application

EXAMPLE

Application program to create a new XFaceMaker which includes the function strncpy.

```
#include <Fm.h>
extern char * strncpy();
main (argc, argv)
int argc;
char **argv; {
Widget toplevel;
toplevel=FmInitiaize("myxfm","Xfm",NULL,0,&argc,argv ) ;
FmAttachFunction("strncpy",strncpy,"String",3,"String",
                                              "String","Int");
FmCallEditor();
}
```

SEE ALSO

FmEditLoop

C NAME / FACE NAME

FmCallValue **FmCallValue**

SYNOPSIS

```
XtArgVal FmCallValue(
        Widget      widget,
        char        *avname,
        int         count,
        ...         )
```

DESCRIPTION

Activate the set script of a per-call active value. This function stores its count last arguments in a structure (starting at field 1), and calls the *set* script of the specified active value with the address of the structure passed as the $ argument for an immediate active value, and as @ otherwise. The function returns the first field of the structure, which is initialized to 0.

Calling this function is similar to executing an instruction of the form:

<widget>:<name>(...);

in a FACE script.

widget is the widget in which the active value whose set script is to be activated has been defined.

avname is the name of the active value whose set script is to be executed.

count is the number of arguments passed in the arguments following. Storage for the arguments is allocated at the time of the call and freed upon completion.

SCOPE

Application, FACE scripts

EXAMPLE

SEE ALSO

FmWriteValue

FmReadValue

C NAME / FACE NAME

C NAME	FACE NAME
FmCatClose	**none**

SYNOPSIS

```
int FmCatClose(
          nl_catd                    i)
```

DESCRIPTION

Close the catalog identified by the catalog descriptor *i*. A value of -1 means “close all catalogs”.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmCatOpen

C NAME FACE NAME

FmCatFind **none**

SYNOPSIS

```
nl_catd FmCatFind(
            char                *name)
```

DESCRIPTION

Return the descriptor of the catalog named name, or -1 if it is not open. Non-GLS machines return -1.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME — FACE NAME

FmCatGetS — **none**

SYNOPSIS

```
String          FmCatGetS(
                int             setn,
                int         msgn,
                char            *dflt)
```

DESCRIPTION

Is equivalent to catgets, except that it looks for the specified message in all message catalogs opened using FmCatOpen. The message catalogs are searched in the reverse order of the opening order. Non-GLS machines return *dflt*.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmCatOpen

C NAME **FACE NAME**

FmCatOpen **none**

SYNOPSIS

```
nl_catd FmCatOpen(
            char        *s1,
            int         i)
```

DESCRIPTION

Perform the equivalent of the ***Load Catalog...*** command.

s1 specifies the catalog name.

i is currently unused and should be set to 0. The return value is the value returned by catopen. As the application opens catalogs with this function, the catalog descriptors are stored by XFaceMaker so that the interface message strings are fetched in the message catalogs that the application has opened. Non-GLS machines return -1.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmCatClose

FmCatGetS

C NAME **FACE NAME**

FmChangeValueAddress **none**

SYNOPSIS

```
int     FmChangeValueAddress (
        Widget        widget,
        char          *avname,
        XtArgVal      value,
        String        type)
```

DESCRIPTION

This function changes the address attached to the specified active value and associated with the specified widget, and calls the *set* script of the active value with the new address passed as the $ argument. If it succeeds, the function returns 1. If the active value does not exist, the function does nothing and returns 0. If *widget* is null, the function returns -1.

Calling this function is similar to executing an instruction of the form:

<widget>:$<name> = value;

in a FACE script.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmAttachValue

C NAME **FACE NAME**

FmClearArgList **FmClearArgList**

SYNOPSIS

```
None   FmClearArgList (
       FmArgList        list)
```

DESCRIPTION

Removes all arguments from a list (but does not free the list itself).

SCOPE

FACE scripts   application

EXAMPLE

Taken from the DragDrop example in the distribution:

```
FmArgList arglist = FmNewArgList();
FmAddArg(arglist,"pixmap",
                      FmGetPixmap(self"letters.bm"));
FmAddArg(arglist,"mask",ConvertString
                      (self,"letters_m.bm","Bitmap"));
Widget source_icon = FmCreateDragIcon(self,
                              "source",arglist);
FmClearArgList(arglist);
```

SEE ALSO

FmFreeArgList

C NAME **FACE NAME**

FmCreateDragIcon **FmCreateDragIcon**

SYNOPSIS

```
Widget    FmCreateDragIcon(
              Widget        widget,
              String        name,
              FmArgList     arglist)
```

DESCRIPTION

Call XmCreateDragIcon with the arguments specified by *arglist*.

widget specifies the ID of the widget where default values for the drag icons's visual attributes should be fetched.

name is the name of the DragIcon widget that will be created.

arglist specifies the argument list.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmCreateManagedObject **none**

SYNOPSIS

```
Widget      FmCreateManagedObject(
            Widget                    parent,
            char                      *object_name,
            char                      *group_name,
            Arg                       *args,
            Cardinal                  nargs)
```

DESCRIPTION

Same as FmCreateObject, but the toplevel object created is mapped or managed. This function has been replaced by FmLoadCreateManagedGroup which is more efficient and offers a more consistent parameter interface.

SCOPE

Application only

EXAMPLE

```
group1=FmCreateManagedObject(top,”group1”,
                                        “group.fm”, NULL, 0);
```

SEE ALSO

FmCreateObject

FmLoadCreateManagedGroup

FmLoadGroup

FmLoadCreateGroup

C NAME ## FACE NAME

FmCreateObject **none**

SYNOPSIS

```
Widget      FmCreateObject(
            Widget      parent,
            char        *object_name,
            char        *group_name,
            Arg         *args,
            Cardinal    nargs)
```

DESCRIPTION

Create an instance of the group named *group_name* every time this function is called. The group must have been previously loaded using FmLoadGroup. This function has been replaced by FmLoadCreateGroup which is more efficient and offers a more consistent parameter interface.

SCOPE

Application only

EXAMPLE

```
group1=FmCreateObject(top,"group1","group.fm",NULL,0);
```

SEE ALSO

FmLoadGroup

FmLoadCreateGroup

FmLoadCreateManagedGroup

C NAME | FACE NAME

FmCreateRectangle | **FmCreateRectangle**

SYNOPSIS

```
XRectangle          FmCreateRectangle(
                    int     x,
                    int     y,
                    int     width,
                    int     height)
```

DESCRIPTION

Allocate and fill an XRectangle structure with the specified geometry. This structure can be passed to FmDropSiteRegister as the value of the resource **XmNdropRectangles**. The pointer returned should be freed when no longer needed, using XtFree.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmCreateXmString **CreateXmString**

SYNOPSIS

```
XmString FmCreateXmString (
            String      s)
```

DESCRIPTION

Create an XmString using the string s and the default character set. When it is no longer needed, the string can be freed using XmStringFree.

SCOPE

FACE scripts application

EXAMPLE

Specify a side effect for the XtSetValues of a new resource of type *String* when defining a class. For example, in the resource's *set* script:

```
xms = CreateXmString (@);
self.PB:labelString = xms;
StringFree (xms);
```

SEE ALSO

FmGetString

XmStringFree

C NAME / FACE NAME

FmDeleteObject **FmDeleteObject**

SYNOPSIS

```
void        FmDeleteObject(
            Widget        w)
```

DESCRIPTION

Destroy Widget *w* and all its subwidgets. The widget must have been created by one of the following: FmLoadCreate, FmLoad, FmLoadCreateGroup, FmLoadCreateManagedGroup, FmCreateObject, or FmCreateManagedObject.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME   FACE NAME

FmDeleteVariable   **none**

SYNOPSIS

```
void FmDeleteVariable(
                String                  name)
```

DESCRIPTION

Delete the global FACE variable *name*. This is useful only when working with interfaces in interpreted mode. FmDeleteVariable is not available in the libFm_c.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmDisableTraversal **FmDisableTraversal**

SYNOPSIS

```
None FmDisableTraversal (
            Widget      w)
```

DESCRIPTION

Disable keyboard traversal for Widget *w* and all its children by calling Xm-RemoveTabGroup(*w*) and setting the resources **traversalOn** to False and **highlightThickness** to 0 for all children of *w*.

SCOPE

FACE scripts application

EXAMPLE

```
FmDisableTraversal(parent.XmRowColumn0);
```

SEE ALSO

FmEnableTraversal

XmProcessTraversal

C NAME / FACE NAME

C NAME	FACE NAME
FmDoEvent	**FmDoEvent**

SYNOPSIS

None FmDoEvent()

DESCRIPTION

Process a single X event. This function calls XtNextEvent and then XtDispatchEvent once.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FmLoop

C NAME FACE NAME

FmDragStart **FmDragStart**

SYNOPSIS

```
Widget      FmDragStart(
                Widget      widget,
                XEvent      *event,
                String      targets,
                FmArgList   arglist)
```

DESCRIPTION

Call XmDragStart with the arguments specified by *arglist*. The XmNexport-Targets and XmNnumExportTargets arguments are specified as a comma-separated string in *targets*.

SCOPE

FACE scripts application

EXAMPLE

```
FmDragStart(self,$,”STRING”,arglist);
```

SEE ALSO

FmStartSimpleDrag

C NAME FACE NAME

FmDropSiteRegister **FmDropSiteRegister**

SYNOPSIS

```
Widget      FmDropSiteRegister(
            Widget      widget,
            String      targets,
            FmArgList   arglist
```

DESCRIPTION

Call XmDropSiteRegister with the arguments specified by *arglist*. The Xm-NimportTargets and XmNnumImportTargets arguments are specified as a comma-separated string in *targets*.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FmRegisterSimpleDropSite

C NAME | FACE NAME

FmDropSiteRetrieve | **FmDropSiteRetrieve**

SYNOPSIS

```
Any          FmDropSiteRetrieve(
             Widget       widget,
             String       name)
```

DESCRIPTION

Call XmDropSiteRetrieve to retrieve the value of the resource *name*, and return the retrieved value.

widget is the ID of the widget that encloses the drop site.

name is the name of the resource to be retrieved.

The value of the resource is returned by the function. Only one resource can be retrieved for each call.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

XmDropSiteRetrieve

C NAME | FACE NAME

FmDropSiteUpdate **FmDropSiteUpdate**

SYNOPSIS

```
Widget      FmDropSiteUpdate(
            Widget      widget,
            String      name,
            None        value)
```

DESCRIPTION

Call XmDropSiteUpdate with an *Arg* built using *name* and *value*.

widget is the ID of the widget registered as a drop site where the resource is to be changed.

name is the name of the resource to be changed.

value is the new value for the resource.

SCOPE

FACE scripts, application

EXAMPLE

SEE ALSO

XmDropSiteUpdate

C NAME   FACE NAME

FmDropTransferAdd   **FmDropTransferAdd**

SYNOPSIS

```
Any         FmDropTransferAdd(
            Widget      widget,
            Widget      drop_transfer,
            String      targets,
            None        client_data)
```

DESCRIPTION

Call XmDropTransferAdd with a drop transfer entry list built using the *targets* and *client_data* arguments.

widget is the ID of the widget in which the drop will occur.

drop_transfer is the ID of the Drop Transfer widget returned by XmDropTransferStart.

targets is a comma-separated list of targets that are to be added to the drop targets list.

client_data is the client data you want to pass tp the drop site's active value. The set script can retrieve it from the FmTransferProcStruct structure.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

C NAME

FmDropTransferStart

FACE NAME

FmDropTransferStart

SYNOPSIS

```
Any   FmDropTransferStart(
      Widget      widget,
      Widget      drag_context,
      String      targets,
      None        client_data,
      Boolean     incremental,
      Int         status)
```

DESCRIPTION

Call XmDropTransferStart with the specified *drag_context*. The value of the XmNdropTransfers resource is built using the *targets* and *client_data* arguments. The *incremental* and *status* arguments are passed as the XmNincremental and XmNtransferStatus resources respectively.

widget is the ID of the widget in which the drop will occur.

drag_context is the ID of the Drag Context widget associated to the transaction.

targets is a comma-separated list of targets.

client_data is the client data you want to pass to the drop site's active value. The set script can retrieve it from the FmTransferProcStruct structure.

incremental is the value stored in the XmNincremental resource of the Drop Transfer widget.

status is the value stored in the XmNtransferStatus resource of the Drop Transfer widget.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

XmDropTransferStart

C NAME | FACE NAME

FmDtoi **dtoi**

SYNOPSIS

```
Int        FmDtoi(
           Pointer      dp)
```

DESCRIPTION

Return the conversion of the *double* pointed to by *dp* into an integer.

SCOPE

FACE scripts   application

EXAMPLE

```
parent.scale:value = dtoi(dblval);
```

SEE ALSO

FmItod

atoi

C NAME **FACE NAME**

FmEditLoop **none**

SYNOPSIS

void FmEditLoop()

DESCRIPTION

The same as FmLoop but allows the user to load the editor using the CallEditor function by typing <Shift-Control-Escape> on the keyboard while the pointer is in any application window other than one which was popped up by the function PopupAndWait.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmLoop

C NAME FACE NAME

FmEnableTraversal **FmEnableTraversal**

SYNOPSIS

```
None        FmEnableTraversal (
            Widget      w,
            Int         ht)
```

DESCRIPTION

Enable keyboard traversal for Widget *w* and all its children by calling XmAddTabGroup(*w*) and setting the resources traversalOn to True and highlightThickness to *ht* for all children of *w*.

SCOPE

FACE scripts application

EXAMPLE

```
EnableTraversal(parent.XmRowColumn2);
```

SEE ALSO

FmDisableTraversal

XmProcessTraversal

C NAME FACE NAME

FmEqualString **FmEqualString, equal**

SYNOPSIS

```
Boolean     FmEqualstring(
            String      s1,
            String      s2)
```

DESCRIPTION

Return *true* if strings *s1* and *s2* are identical.

SCOPE

FACE scripts application

EXAMPLE

Compare an active value called File with the empty string:

```
GetActiveValue(parent.TextFile,"File");
if(equal( $File, "")){
Show(root.FileNameHelp);
}
```

SEE ALSO

not_equal

strcmp

C NAME | FACE NAME

FmFetchValue | **none**

SYNOPSIS

```
Boolean     FmFetchValue (
            Widget        widget,
            char          *name,
            XtPointer     *address_return,
            XrmQuark      *type_return,
            int           *storage_return,
            int           *scope_return,
            int           *immediate_return,
            int           *automatic_return,
            int           *genfun_return)
```

DESCRIPTION

This function returns the attributes of a given active value for a given object. If the active value is automatic, not immediate and it is not yet attached, the active value storage is allocated. The contents of *storage_return* is one of the constants FM_AV_GLOBAL, FM_AV_OBJECT or FM_AV_NONE defined in Fm.h. The contents of *scope_return* is one of the constants FM_AV_PUBLIC, FM_AV_PRIVATE, or FM_AV_PROTECTED defined in Fm.h. *immediate_return*, *automatic_return,* and *genfun_return* contain 0 or 1. The function returns True if the active value exists, False otherwise.This is the new version of **FmGetActiveValueAddr**, and should be used instead of it.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmGetActiveValueAddr

C NAME | FACE NAME

FmFetchValueAddress **none**

SYNOPSIS

```
int     FmFetchValueAddress(
        Widget        widget,
        char          *name,
        XtArgVal      *value_return,
        String        *type_return)
```

DESCRIPTION

This function fetches the address attached to the specified active value and associated with the specified widget, calls the *get* script of the active value with the address passed as the $ argument, and returns the address. The type returned is the type of the active value if it is immediate, or Pointer otherwise. If it succeeds, the function returns 1. If the active value does not exist, the function returns 0. If *widget* is null, the function returns -1.

Calling this function is similar to executing an instruction of the form:

value = <widget>:$<name>;

in a FACE script.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmFetchValue

C NAME   FACE NAME

FmFreeArgList   **FmFreeArgList**

SYNOPSIS

```
None        FmFreeArgList (
            FmArgList    list)
```

DESCRIPTION

Remove all the arguments from a list and free the list itself. The list must not be used again.

SCOPE

FACE scripts   application

EXAMPLE

```
FmFreeArgList(arglist);
```

SEE ALSO

FmClearArgList

FmNewArgList

FmAddArg

C NAME FACE NAME

FmFtoi **ftoi**

SYNOPSIS

```
Int          FmFtoi(
             Pointer      fp)
```

DESCRIPTION

Return the result of converting a float pointed to by *fp* into an integer.

SCOPE

FACE scripts application

EXAMPLE

Print the decimal value of a Float active value called fval:

printf("decimal value of fval: %d\n”, ftoi($fval));

SEE ALSO

FmItof

C NAME　FACE NAME

FmGetActiveValue　**GetActiveValue**

SYNOPSIS

```
Boolean     FmGetActiveValue (
            Widget      w,
            String      name)
```

DESCRIPTION

Call the get script of active value *name* for Widget *w*. If *name* is null, call the get scripts for all active values defined for *w* regardless of their names. If *w* is null, call the get scripts for all active values called *name* regardless of the Widget they are defined for. If at least one script was called, return 1, otherwise, return 0.

SCOPE

FACE scripts, application

EXAMPLE

Activate the get script for active value "fval" in widget parent.SB:

GetActiveValue(parent.SB,"fval");

Activate the get script for active value "fval" in all widgets:

GetActiveValue(Widget(0),"fval");

SEE ALSO

FmSetActiveValue

FmGetActiveValueTimeOut

C NAME **FACE NAME**

FmGetActiveValueAddr **GetActiveValueAddr**

SYNOPSIS

```
String        FmGetActiveValueAddr (
              String          name)
```

DESCRIPTION

Return the address attached to the active value named *name*, or 0 if no address was attached to *name*.

SCOPE

FACE scripts application

EXAMPLE

Print the value of a float active value:

```
ptr = GetActiveValueAddr("fval");
printf("decimal val:%d\n”, ftoi(Pointer(ptr)));
```

SEE ALSO

FmAttachAv

FmFetchValueAddress

FmAttachValue

C NAME FACE NAME

FmGetActiveValueTimeOut **GetActiveValueTimeOut**

SYNOPSIS

```
TimeOutClosure    FmGetActiveValueTimeOut(
                  Widget       w,
                  String       name,
                  Int          interval)
```

DESCRIPTION

Initialize a timeout so that the get script of the active value *name* is called in *interval* milliseconds by: GetActiveValue(*w*, *name*). It returns a pointer to a timeout identifier TimeoutID. The timeout can be stopped with StopTime-Out(Pointer TimeoutID). The *w* parameter must *not* be null, but *name* can be.

SCOPE

FACE scripts, application

EXAMPLE

Definition of an active value which re-activates its get script every second:

```
name = Myav
get = global handle;
  Beep();
  handle = GetActiveValueTimeOut(self,"Myav",1000);
```

a push Button which activates the active value's get script:

```
labelString = Get_activeval
activateCallback = global handle;
handle = GetActiveValueTimeOut(parent.avwidget,"Myav",1000);
```

a pushButton which stops the process:

```
labelString = Stop_activeval
activateCallback = global handle; StopTimeOut(handle);
```

SEE ALSO

FmSetActiveValueTimeOut

FmStopTimeOut

C NAME FACE NAME

FmGetCatD **none**

SYNOPSIS

```
nl_catd FmGetCatD()
```

DESCRIPTION

Return the descriptor of the last opened catalog. Non-GLS machines will return -1.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmCatOpen

FmCatGetS

C NAME FACE NAME

FmGetColorPixmap **none**

SYNOPSIS

```
Pixmap      FmGetColorPixmap(
            Screen      screen,
            char        *image_name,
            Pixel       foreground,
            Pixel       background)
```

DESCRIPTION

Similar to XmGetPixmap but can also load color Pixmap files in the XWD format (used by the xwd and xwud commands). The function first calls XmGetPixmap to see if the image is already loaded or if it is a bitmap file name, in which case the bitmap is loaded. Otherwise, it searches first in the directories defined by FMPIXMAPSPATH with a substitution character of %P followed by XBMLANGPATH with a substitution character of %B for the XWD file *image_name*. If found, the pixmap is installed and the Pixmap returned; if not, the value XmUNSPECIFIED_PIXMAP is returned.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmGetPixmap

C NAME   FACE NAME

FmGetFunctionsVector   **none**

SYNOPSIS

FmFunctionsVector FmGetFunctionsVector()

DESCRIPTION

Register a new widget class without directly referencing Fm functions. This function returns a structure containing pointers to XFaceMaker functions used to register new widget classes. When this function is used, the Fm functions are referenced through function pointers rather than directly by the object file of the class so that applications using the widget need not be linked with the Fm_c library.

SCOPE

Application only

EXAMPLE

FmAddFooWidgetClass(FmGetFunctionsVector())

SEE ALSO

FmAddWidgetClass

C NAME FACE NAME

FmGetPixmap **FmGetPixmap**

SYNOPSIS

```
Pixmap      FmGetPixmap(
            Widget      w,
            char        image_name)
```

DESCRIPTION

Get the **screen**, **foreground** and **background** resources for Widget *w* and call XmGetPixmap.

SCOPE

Application, FACE scripts

EXAMPLE

```
parent.PB:labelType = pixmap;
parent.PB:labelPixmap = FmGetPixmap(self,”scales”);
```

SEE ALSO

FmGetColorPixmap

C NAME / FACE NAME

FmGetString **GetString**

SYNOPSIS

```
String          FmGetString(
                XmString            str)
```

DESCRIPTION

Return an ordinary zero terminated character string equal to the text contained within the XmString *str* assuming XmSTRING_DEFAULT_CHARSET. Calls XmStringGetLtoR. When it is no longer needed, the string can be freed using free.

SCOPE

FACE scripts application

EXAMPLE

Declare and initialize a FACE array to contain the items of an XmList widget; print its contents:

```
XmString a[Int];
count = parent.XmList0:itemCount;
a = new_c_array(Pointer(parent.XmList0:items),count);
for (i=0; i < count; i=i+1)
printf("index: %d item %s\n",i,FmGetString(a[i]));
```

SEE ALSO

FmCreateXmString

C NAME — FACE NAME

FmGetStringFromTable — **FmGetStringFromTable**

SYNOPSIS

```
String          FmGetStringFromTable(
                XmStringTable               table)
```

DESCRIPTION

FmGetStringFromTable returns the first string in an XmStringTable.

table is the XmStringTable.

SCOPE

FACE scripts   application

EXAMPLE

```
parent.TF:value =
        FmGetStringFromTable(parent.Li:items);
```

SEE ALSO

C NAME / FACE NAME

FmGetValue **FmGetValue**

SYNOPSIS

```
Boolean     FmGetValue (
            Widget       widget,
            char         *name,
            XtPointer    address)
```

DESCRIPTION

This function calls the get script of the specified active value for the specified widget, passing it the specified address as the $ argument. This function is similar to FmGetActiveValue, but allows you to specify the attached address for each call.

widget is the widget in which the active value has been defined.

name is the name of the active value.

address is the address to be passed as $ to the active value get script.

Returns True if the get script is executed, False otherwise.

SCOPE

FACE scripts, application

EXAMPLE

SEE ALSO

FmGetActiveValue

FmAttachValue

C NAME FACE NAME

FmGetVersionString **GetVersionString**

SYNOPSIS

String FmGetVersionString()

DESCRIPTION

Return a string describing the version of XFaceMaker, such as "Version 3.0". When it is no longer needed, the string can be freed using free.

SCOPE

FACE scripts application

EXAMPLE

Print the version of XFaceMaker:

```
tmp = GetVersionString();
printf("%s\n”,tmp);
free(tmp);
```

SEE ALSO

C NAME	**FACE NAME**
FmGetXmDisplay	**FmGetXmDisplay**

SYNOPSIS

```
Widget      FmGetXmDisplay(
            Widget      w)
```

DESCRIPTION

Call XmGetXmDisplay with widget *w*'s display, and return the global Xm-Display widget id.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmGetXmScreen **FmGetXmScreen**

SYNOPSIS

```
Widget      FmGetXmScreen(
            Widget      w)
```

DESCRIPTION

Call XmGetXmScreen with Widget *w*'s display, and return the global Xm-Screen widget id.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

FmHideWidget **FmHideWidget, Hide, hide**

SYNOPSIS

```
None        FmHideWidget(
            Widget      w)
```

DESCRIPTION

Hide widget *w* from the screen. If *w* is a Shell widget, then Hide unmaps the Shell. If *w* is a Popup-Shell, then Hide calls XtPopdown on the shell. Otherwise, Hide simply unmanages the Widget. Note: if the intent is to hide an Xm-DialogShell, *w* should refer to the DialogShell's child, not to the shell itself because of the way Motif handles DialogShells.

SCOPE

FACE scripts	application

EXAMPLE

Hide a transientShell which is the shell ancestor of self:

Hide(toplevel);

SEE ALSO

FmShowWidget

FmLowerWidget

C NAME	**FACE NAME**
FmInitialize	**none**
FmInitializeC	**none**
FmInitializeI	**none**

SYNOPSIS

```
Widget    FmInitialize(
          String              name,
          String              class,
          XrmOptionDescRec    ops[],
          Cardinal            num_ops,
          Cardinal            *argc,
          String              argv[])
```

DESCRIPTION

Initialize all data structures required by XFaceMaker, and call XtInitialize. The return value is the default ApplicationShell returned by XtInitialize. You must call FmInitialize at the beginning of the application, not XtInitialize, unless you have generated the c code interface with the standc flag set. FmInitialize has the same arguments as XtInitialize and you should consult the X documentation for a full explanation. The line:

```
toplevel=FmInitialize("name","Name",NULL,0,&argc,argv);
```

is sufficient for an application called *name* which does not need to parse the command line to recognize its own options. If you pass the class Xfm to FmAppInitialize, it will be replaced by XfmApplication. Otherwise, the class name that you pass will be the class name of the application you are building.

The FmInitialize function is also available in two versions:

FmInitializeI is defined in the libFm_e and libFm libraries, while FmInitializeC is defined in libFm_c. The arguments and return values are the same as for FmInitialize. See "Compatibility with libFm_c and lib_Fm_e" in the Application chapter of the User's Guide.

SCOPE

Application only

SEE ALSO

XtInitialize

FmAppInitialize

C NAME | FACE NAME

FmInternationalize | **Internationalize**

SYNOPSIS

```
String FmInternationalize(
                    String          str)
```

DESCRIPTION

Scan the string *str* which must be of the form %MSG%n%m xxx, call FmCat-GetS, searching the message catalogs that are currently open, and return the resulting string, or NULL if *str* does not have the correct format.

SCOPE

FACE scripts application

EXAMPLE

Internationalize a string returned by an application function:

```
function String GetMyString();
s1 = GetMyString();
s2 = Internationalize(s1);
```

SEE ALSO

C NAME — FACE NAME

FmItoa — **itoa**

SYNOPSIS

```
String      FmItoa(
            Int     i)
```

DESCRIPTION

Convert integer *i* into a statically allocated string.

SCOPE

FACE scripts	application

EXAMPLE

Set a push button’s label string to the value of a scale:

```
s = itoa(parent.scale:value);
parent.GetLabel:labelString = s
free(s);
```

SEE ALSO

FmItod

atoi

C NAME FACE NAME

FmItod **itod**

SYNOPSIS

None	FmItod(	
	Int	*i*,
	Pointer	*dp*)

DESCRIPTION

Convert integer *i* into a double and store the result in the double variable pointed to by *dp*.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

FmItoa

FmDtoi

C NAME FACE NAME

FmItof **itof**

SYNOPSIS

```
None        FmItof(
            Int         i,
            Pointer     fp)
```

DESCRIPTION

Convert integer *i* into a float and store the result in the float variable pointed to by *fp*.

SCOPE

FACE scripts application

EXAMPLE

Store the value of a scale widget in a float active value called fval:

```
itof(parent.scale:value,$fval);
```

SEE ALSO

citof

C NAME	**FACE NAME**
FmListAllowKeySelection	**none**

SYNOPSIS

```
None        FmListAllowKeySelection(
            Widget      list,
            Boolean     alnum,
            String      delim,
            Boolean     notify)
```

DESCRIPTION

FmListAllowKeySelection allows the user to select items in the list by typing characters on the keyboard. For each non-blank key typed in the list while it has keyboard focus, the list selects the next item in the list that starts with that character.

list is the XmList widget id.

alnum is a flag indicating whether to accept non-alphanumeric characters. If *alnum* is true, only letters and digits will cause selection to take place. If false, then the user may select with any non-blank character.

delim is a string containing delimiters that mark the search area for selection. Each item will be searched after the last delimiter in that item's value. For example, if you specify "/" as the delimiter in a list of filenames, then only the last component determines the result of the search.

notify indicates whether or not the list should call its selection callback list when a selection is made.

SCOPE

application only

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

FmListGetItems **FmListGetItems**

SYNOPSIS

```
String        FmListGetItems(
              Widget        list)
```

DESCRIPTION

Return the items in a list as a string of comma-separated items.

list is the XmList widget ID.

SCOPE

FACE scripts   application

EXAMPLE

```
str = FmListGetItems(parent.Li);
printf(" items %s\n",str);
free(str);
```

SEE ALSO

FmListSetItems

C NAME / FACE NAME

C NAME	FACE NAME
FmListGetNthItem	**ListGetNthItem**

SYNOPSIS

```
String      FmListGetNthItem(
            Widget      list,
            Int         position)
```

DESCRIPTION

A convenience function used to get the n-th item of an XmList widget, where *position* is: last=0, first=1, second=2, etc. Returns a pointer to a character string which should be freed when no longer needed, or NULL if the item is not found.

SCOPE

FACE scripts   application

EXAMPLE

Print the content of a list widget from a sibling pushbutton:

```
activateCallback = w = parent.XmList0;
for (i=1; i <= w:itemCount; i=i+1) {
str = ListGetNthItem(w,i);
printf("%s\n",str);
free(str);
}
```

SEE ALSO

FmListGetNthSelectedItem

C NAME — FACE NAME

FmListGetNthSelectedItem — **ListGetNthSelectedItem**

SYNOPSIS

```
String      FmListGetNthSelectedItem(
            Widget      list,
            Int         position)
```

DESCRIPTION

A convenience function used to get the n-th *selected* item of an XmList widget, where *position* is: last=0, first=1, second=2, selected item etc. It returns a pointer to the character string which should be freed when no longer needed, or NULL if the item is not found.

SCOPE

FACE scripts   application

EXAMPLE

Print the selected items of a list from a sibling pushbutton:

```
activateCallback = w = parent.XmList0;
for (i = 1; i <= w:selectedItemCount; i=i+1) {
str = ListGetNthSelectedItem(w,i);
printf(" selected item number %d, item
                                        %s\n”,i,str);
free(str);
}
```

SEE ALSO

FmListGetNthItem

C NAME FACE NAME

FmListGetSelectedItems **FmListGetSelectedItems**

SYNOPSIS

```
String        FmListGetSelectedItems(
              Widget        list)
```

DESCRIPTION

Return the selected items in a list as a string of comma-separated items.

list is the XmList widget.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FmListGetNthSelectedItem

C NAME	**FACE NAME**
FmListSetItems	**ListSetItems**

SYNOPSIS

```
None        ListSetItems(
            Widget      list,
            String      items)
```

DESCRIPTION

A convenience function used to set the *items* and *itemCount* resources of the XmList widget, according to the *items* string. *items* is a string containing comma-separated items. ListSetItems converts this string to an XmString-Table, and calls XtSetValues to set the items and itemCount resources.

SCOPE

FACE scripts   application

EXAMPLE

Initialize a list in its create callback:

```
xfmCreateCallback = ListSetItems(self,"first,second,third,fourth,
                                        fifth,sixth");
```

SEE ALSO

FmListGetItems

C NAME **FACE NAME**

FmLoad **none**

SYNOPSIS

```
Widget          FmLoad(
                char    *filename)
```

DESCRIPTION

Load, create and manage the interface contained in *filename*, where *filename* is a .fm file saved by the ***Save*** or ***Save As*** commands of XFaceMaker. *filename* must normally contain an ApplicationShell, which in turn contains the other widgets of your interface (and possibly additional TopLevelshells if your application has several windows). If the topmost widget of *filename* is not an ApplicationShell, then it must be of a subclass of Shell, and it will be created as a child of the default ApplicationShell returned by FmInitialize. In this case, the default Applicationshell will be visible on the display as a small window in the top right hand corner containing the current version of XFaceMaker.

FmLoad returns the topmost widget of your interface, normally an ApplicationShell, or if there are several main shells[1], the first one. If it is an ApplicationShell this will replace the default one created by FmInitialize.

Notes: If your interface contains more than one window, then you should define one of these windows as the *main* window, and the additional windows as the *secondary* windows. The main window should be an ApplicationShell, and secondary windows should be TopLevelShells with their popup flag set.

If your interface scripts call application functions, you will need to attach them *before* you load the interface. See FmAttachFunction.

SCOPE

Application only

1. "main shells" are those that are not children of one another. NSL does not recommend building files with more than one toplevel object since only the first toplevel object will be accessible through the widget ID returned by FmLoad. Note that, while FmLoad does load all the hierarchies deriving from multiple toplevel shells, this is not the case with FmLoadCreate, FmLoadCreateGroup, FmLoadCreateManagedGroup, and FmLoadGroup.

EXAMPLE

SEE ALSO

FmLoadCreate

FmLoadCreateManaged

FmLoadCreateManagedGroup

C NAME — FACE NAME

FmLoadCreate — **none**

SYNOPSIS

```
Widget      FmLoadCreate(
            char         *name,
            char         *filename,
            Widget       parent,
            Arg          *args,
            Cardinal     *nargs)
```

DESCRIPTION

Load and create the fm interface contained in *filename*. Unlike FmLoad, does not manage the interface, even if the interface was saved as visible. Unlike FmLoad, if the interface contains several main shells, only the first one will be loaded and created. You can specify a parent and creation arguments. The toplevel object created is returned. FmLoadCreate always reads in the specified .fm file and creates the widget hierarchy each time it is called. It does not save an internal description in memory and should therefore be used whenever a widget hierarchy is to be created only once.

Use FmLoadCreate to create interface fragments that do not contain an ApplicationShell.

The parameters are:

name is the name under which the topmost widget of the hierarchy is created. If you need to create several instances of the interface with different names you should use FmLoadCreateGroup and change this parameter for each instance.

filename is the name of the .fm file which is to be loaded.

parent specifies the widget of which the topmost widget will become a child. If parent is NULL, and the topmost widget is not an ApplicationShell, then the instance is created as a child of the default ApplicationShell returned by FmInitialize. However, it is better to do this explicitly as shown in the examples above.

args, nargs are additional resource values that will be passed to XtCreate-Widget at the creation of the topmost widget. *args* and *nargs* override any values specified within the .fm file. Typically, these are used to specify the position and size of the instance. You should pass NULL and 0 if you do not want to change any resources.

FmLoadCreate always reads in the specified .fm file and creates the widget hierarchy each time it is called. It does not save an internal description in memory and should therefore be used whenever a widget hierarchy is to be created only once.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmLoad

FmLoadCreateManaged

FmLoadCreateManagedGroup

C NAME   FACE NAME

FmLoadCreateGroup   **none**

SYNOPSIS

```
Widget      FmLoadCreateGroup(
            char          *name,
            char          *filename,
            Widget        parent,
            Arg           *args,
            Cardinal      *nargs)
```

DESCRIPTION

Checks if the .fm file *filename* is already loaded, and if not, loads it by calling FmLoadGroup(*name, filename*). Then, an instance of the group is created as a child of *parent* using the creation arguments specified by *args* and *nargs*. The toplevel object created is returned. Does not manage the group, even if it was saved as visible. The parameters are the same as those described above for FmLoadCreate. Unlike FmLoad, if the interface contains several main shells, only the first one will be loaded and created.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmLoadCreateManagedGroup

C NAME   FACE NAME

FmLoadCreateManaged **none**

SYNOPSIS

```
Widget     FmLoadCreateManaged (
           char        *name,
           char        *filename,
           Widget      parent,
           Arg         *args,
           Cardinal    *nargs)
```

DESCRIPTION

Same as FmLoadCreate, but the top level object created is managed or mapped even if the interface is saved as hidden. Unlike FmLoad, if the interface contains several main shells, only the first one will be loaded and created.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmLoad

FmLoadCreate

FmLoadCreateManagedGroup

C NAME FACE NAME

FmLoadCreateManagedGroup **none**

SYNOPSIS

```
Widget      FmLoadCreateManagedGroup(
            char          *name,
            char          *filename,
            Widget        parent,
            Arg           *args,
            Cardinal      *nargs)
```

DESCRIPTION

Same as FmLoadCreateGroup, but the top level object created is managed or mapped even if it was saved as hidden. Unlike FmLoad, if the interface contains several main shells, only the first one will be loaded and created.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmLoadGroup

FmLoadCreateGroup

C NAME — FACE NAME

FmLoadGroup — **LoadGroup**

SYNOPSIS

```
int        FmLoadGroup(
               char    *group_name,
               char    *group_file)
```

DESCRIPTION

Load a widget group from the interface file *group_file*. The group is named *group_name*. FmLoadGroup returns 0 if the group could not be loaded, and non-zero otherwise. Unlike FmLoad, FmLoadGroup only loads and parses the .fm file. FmCreateObject or FmCreateManagedObject must subsequently be called to create instances of the group. Therefore in most cases the functions FmLoadCreateGroup and FmLoadCreateManagedGroup should be used. Unlike FmLoad, if the interface contains several main shells, only the first one will be loaded and created.

SCOPE

Face script, application

EXAMPLE

SEE ALSO

FmLoadCreateGroup

FmLoadCreateManagedGroup

FmLoad

C NAME FACE NAME

FmLoadWidgetClass **none**

SYNOPSIS

```
WidgetClass        FmLoadWidgetClass(
                   String        classname,
                   String        classfile)
```

DESCRIPTION

Load the .fm file describing a new widget class for use in an application running in interpreted mode.

classname is the name of the class which should be loaded

classfile is the name of the .fm format class description file.

When XFaceMaker generates the C code for the class, it defines a function called FmAdd<ClassName>WidgetClass which can be used to include the class into XFaceMaker. The function is controlled by the XFM compilation flag.

SCOPE

Application only

EXAMPLE

An application loading an interface called myinterface.fm which makes use of a new widget class Myclass whose definition is in file myclass.fm would do the following:

```
w=FmInitialize(“myappli”,”Test”,NULL,0,&argc,argv);
k = FmLoadWidgetClass(“Myclass”,”myclass.fm”);

/*attach functions and active values here*/

ww = FmLoad(“interface.fm”);
FmLoop();
```

SEE ALSO

C NAME FACE NAME

FmLoop **none**

SYNOPSIS

void FmLoop()

DESCRIPTION

Pass control to the function XtAppMainLoop. This function must be called after the initialization phase of your application to handle X events.

If you are generating standalone code and do not want to link with the Fm_c library, call XtAppMainLoop (or XtMainLoop) instead.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

FmLowerWidget **LowerWidget**

SYNOPSIS

```
None            FmLowerWidget(
                Widget          w)
```

DESCRIPTION

Call XLowerWindow on *w*'s window.

SCOPE

FACE scripts application

EXAMPLE

Find a widget called List0 starting from the top of the interface hierarchy and lower it:

```
w = XtNameToWidget(root,"*List0");
LowerWidget(w);
```

SEE ALSO

FmRaiseWidget

FmHideWidget

C NAME — FACE NAME

FmNewArgList — **NewArgList, FmNewArgList**

SYNOPSIS

FmArgList FmNewArgList ()

DESCRIPTION

Allocate an empty argument list.

SCOPE

FACE scripts   application

EXAMPLE

```
FmArgList arglist = FmNewArgList();
FmAddArg(arglist,"pixmap",
        FmGetPixmap(self"letters.bm"));
FmAddArg(arglist,"mask",ConvertString
                (self,"letters_m.bm","Bitmap"));
Widget source_icon = FmCreateDragIcon(self,
                        "source",arglist);
FmClearArgList(arglist);
```

SEE ALSO

FmClearArgList

FmFreeArgList

FmAddArg

C NAME — FACE NAME

FmNewPredefinedVariable — **none**

SYNOPSIS

```
int     FmNewPredefinedVariable(
        String        name,
        XtArgVal      value,
        String        type)
```

DESCRIPTION

Declares a new predefined FACE variable. Its effect is equivalent to typing define <*name*> <*value*> in a FACE script, but it allows you to specify a type other than Int or String. This function is useful for declaring pre–defined values for a new resource type.

SCOPE

Application only

EXAMPLE

Define a Float value:

```
static XtArgVal _ftov_(
#if NeedFunctionPrototypes
float f)
#else
f)
float f;
#endif
{       float _f = f;
        XtArgVal v;
        *(float*)&v = _f;
        return(v);
}
FmNewPredefinedVariable("UNDEFINED",
                _ftov_(UNDEFINED_FLOAT), "Float");
```

SEE ALSO

C NAME   FACE NAME

FmNewType **none**

SYNOPSIS

```
void FmNewType(
          String   type)
```

DESCRIPTION

Declare a new predefined FACE type. Its effect is equivalent to typing type <*type*> in a FACE script.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

FmNewTypeDef **none**

SYNOPSIS

```
int     FmNewTypeDef(
        String        type,
        String        ctype,
        int           size)
```

DESCRIPTION

Has an effect similar to the FACE keyword type. Useful when you want to build a new instance of XFaceMaker with additional built-in functions. Is equivalent to the new “type” construct:

```
FmNewTypeDef
("MyType","struct my_type
        *",sizeof(structmy_type*));
```

The third argument specifies the size of the C type in bytes. This is not used for C/C++ prototyping, but for widget class generation. If the given type is used as the type of a new resource, XFaceMaker will take the specified size as the size of the resource.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmNotEqualString **not_equal**

SYNOPSIS

```
Boolean     FmNotEqualString(
            String      s1,
            String      s2)
```

DESCRIPTION

Returns true if strings *s1* and *s2* are different.

SCOPE

FACE scripts application

EXAMPLE

Print the string returned by the callbacks of a message box if it is not Cancel.

```
str=PopupAndWait
(parent.parent.XmDialogShell0.XmMessageBox0);
if (not_equal(str,"Cancel"))
printf("%s\n",str);
```

SEE ALSO

FmEqualString

strcmp

C NAME FACE NAME

FmPopupAndWait **PopupAndWait**

SYNOPSIS

```
String      FmPopupAndWait(
            Widget      w)
```

DESCRIPTION

Pops up the widget *w* and the program enters a local event-handling loop, until the Return function is called. The program then exits the local loop, and returns the value passed as an argument to Return. If the popup flag is set in the Resources Box, and the shell is a TransientShell, the shell grabs all events, thus forcing the user to answer the popup box.

Note: All possible return points must use Return to ensure the local event loop is exited properly. The return value should not be freed if the argument passed to Return is statically allocated.

SCOPE

FACE scripts

EXAMPLE

In a Pushbutton, show a selectionBox, print the string returned by the selectionBox pushbutton callbacks, and change the labelString:

```
ret = PopupAndWait(root.FareDB.FareSB);
printf("ret= %s\n”,ret);
parent.Fare:labelString = ret;
```

SEE ALSO

FmShowWidget

FmWaitForReturn

FmReturn

C NAME **FACE NAME**

FmRaiseWidget **RaiseWidget**

SYNOPSIS

```
None        FmRaiseWidget(
            widget      w)
```

DESCRIPTION

Call XRaiseWindow on *w'*s X Window.

SCOPE

FACE scripts application

EXAMPLE

Find a widget called List0 starting from the top of the interface hierarchy and raise it:

```
w = XtNameToWidget(root,"*List0");
RaiseWidget(w);
```

SEE ALSO

FmLowerWidget

FmShowWidget

C NAME FACE NAME

FmReadValue **none**

SYNOPSIS

```
int     FmReadValue (
        Widget      widget ,
        char        *name,
        XtArgVal    *value_return,
        String      *type_return)
```

DESCRIPTION

This function fetches the address attached to the specified active value and associated with the specified widget (possibly allocating a memory location of size sizeof(XtArgVal) if the active value is automatic), calls the get script of the active value with the address passed as the $ argument, and returns the contents of the address. The type returned is the type of the active value. If it succeeds, the function returns 1. If the active value does not exist, or if it is not attached and it is not automatic, the function returns 0. If it is immediate, or if *widget* is null, the function returns -1.

widget is the widget ID of the widget in which the active value is defined.

name is the name of the active value.

value_return is the value returned in @ by the get script of the active value

type_return is the type of the @ variable.

Calling this function is similar to executing an instruction of the form:

value = <widget>:@<name>;

in a FACE script.

SCOPE

Application

EXAMPLE

SEE ALSO

FmWriteValue

C NAME | FACE NAME

FmRegisterSimpleDropSite	**FmRegisterSimpleDropSite**

SYNOPSIS

```
None        FmRegisterSimpleDropSite(
            Widget      widget,
            String      targets)
```

DESCRIPTION

Register *widget* as a drop site, which accepts the targets specified in the *targets* argument. If several targets are specified, they must be separated by commas. This function is generally called from a widget's xfmCreateCallback at widget creation time. Similar to FmDropSiteRegister but does not have the *arglist* argument: all arguments other than XmNimportTargets and XmNnumImportTargets have their default values.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

FmDropSiteRegister

C NAME **FACE NAME**

FmRegisterStructures **none**

SYNOPSIS

```
int     FmRegisterStructures(
        FmStructDesc       *structs,
        int                num_structs)
```

DESCRIPTION

Declare the structures described by *structs*, which points to the following type:

```
typedef struct _FmStructDesc} {
        String          name;
        String          cname;
        FmFieldDesc     *fields;
        int             num_fields;
        String          id_field;
        XtArgVal        id_value;
} FmStructDesc;
```

name is the name of the structure; i.e. the FACE type which will be used to declare pointers to the structure. This can be different from the actual C identifier.

cname is the C name which will be used to generate references to the structure in the C code generated by XFaceMaker. It is the string that would be typed in a C file to declare a pointer to the structure.

fields is an array of FmFieldDesc structures:

```
        typedef struct _FmFieldDesc{
                String field_name;
                String field_type;
                int field_size;
                int field_offset;
        }FmFieldDesc;
```

where:

field_name is the name of the field (this must be the same as the C identifier for the structure field);

field_type is the FACE type of the field;

field_size is the size in bytes of the field;

field_offset is the offset in bytes of the field from the beginning of the structure.

num_fields is the number of elements in *fields*.

id_field is currently unused, and must be set to 0.

id_value is currently unused, and must be set to 0.

The offset of a structure field can be computed using the XtOffset macro contained in X11/Intrinsics.h. The **size** can be computed using the FmSize macro, which accepts the same arguments as XtOffset, and which is defined in Fm.h.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmReturn **Return**

SYNOPSIS

```
None        FmReturn(
            Widget      w,
            String      retval)
```

DESCRIPTION

Exit a local event-handling loop entered by a call to FmPopupAndWait, or FmWaitForReturn, and transfer *retval* as a return value. *w* is currently unused and should be set to self.

SCOPE

FACE scripts

EXAMPLE

See PopupAndWait example.

In the SelectionBox applyCallback, return the selected string:

```
str = GetString(self:textString);
Return(self,str);
```

SEE ALSO

FmPopupAndWait

FmWaitForReturn

C NAME — FACE NAME

FmSendClick — **SendClick, send_click**

SYNOPSIS

```
None        FmSendClick(
            Widget      w)
```

DESCRIPTION

Send a *Btn1Down* event followed by a *Btn1Up* event to widget *w*. This simulates a click on the *Select* button.

SCOPE

FACE scripts	application

EXAMPLE

Activate the sibling Cancel button

```
SendClick(parent.Cancel);
```

SEE ALSO

C NAME   FACE NAME

FmSendMessage **SendMessage**

SYNOPSIS

```
Int          FmSendMessage (
             Widget      w,
             String      type,
             String      data)
```

DESCRIPTION

Send a ClientMessage event to Widget *w*. The message_type field of the event is the Atom form of *type*, and the first word of the data field is *data*. The return value is that of the call to the XSendEvent function.

SCOPE

FACE scripts   application

EXAMPLE

A pushButton sends a client message to a label:

```
SendMessage(parent.XmLabel0, "String", "Y");
```

The label receiving the client message changes its labelString to the character passed with SendMessage

```
translations = #override\
<ClientMessage> : eval("XClientMessageEvent ev;\n
ev  = $;\n
buf = '   ';\n
sprintf(buf,'%c',ev->data_b0);\n
self:labelString = buf;")
```

SEE ALSO

XSendEvent

C NAME | FACE NAME

FmSetActiveValue | **SetActiveValue**

SYNOPSIS

```
Boolean     FmSetActiveValue(
            Widget      w,
            String      name)
```

DESCRIPTION

Call the set script of active value *name* for widget *w*. If *name* is null, calls the set scripts of all active values defined for *w* regardless of their names. If *w* is null, calls the set scripts of all active values called *name* regardless of the widget they are defined for. If at least one script was called, it returns 1, otherwise, 0.

SCOPE

FACE scripts   application

EXAMPLE

Activate the set script for active value "fval" in widget parent.SB:

SetActiveValue(parent.SB,"fval");

Activate the set script for active value "fval" in all widgets:

SetActiveValue(Widget(0),"fval");

SEE ALSO

FmGetActiveValue

FmSetActiveValueTimeOut

FmSetValue

C NAME — FACE NAME

FmSetActiveValueTimeOut — **SetActiveValueTimeOut**

SYNOPSIS

```
TimeOutClosure FmSetActiveValueTimeOut(
                        Widget      w,
                        String      name,
                        Int         interval)
```

DESCRIPTION

Initialize a timeout so that the set script of active value *name* is called in *interval* milliseconds by: SetActiveValue(w, name). It returns a handle to a timeout identifier. The timeout can be stopped with FmStopTimeOut to which the handle is passed. Note: The parameter *w* must not be null, but *name* can be.

SCOPE

FACE scripts	application

EXAMPLE

See the example for FmGetActiveValueTimeOut

SEE ALSO

FmStopTimeOut

FmGetActiveValueTimeOut

C NAME **FACE NAME**

FmSetCloseHandler **none**

SYNOPSIS

```
CloseHandPfn        FmSetCloseHandler(
                    CloseHandPfn function)
```

DESCRIPTION

Used to augment or override the default close handler installed by XFace-Maker. The function returns the address of the previously installed close handler function, i.e., that of the default close handler the first time it is called. If the user function returns True, then the default close handler will be called afterwards. If it returns False, the default close handler will not be called.

SCOPE

Application only

EXAMPLE

```
#include<Fm.h>

Boolean Myclose(w)
        Widget          w;
{
        if(XtIsApplicationShell(w)) {
          /* do clean up before exiting: close files, etc */
        } else {
          /* do clean up before unmap */
        }
        return True;
}
main (argc, argv)
int argc;
char **argv;
{
 Widget def_app_sh, app_sh ;
 def_app_sh=FmInitialize("app","App",NULL,0,&argc,argv);
 app_sh = FmCreateinterface("app",def_app_sh,NULL,0);
 FmSetCloseHandler(Myclose);
 FmLoop();
}
```

SEE ALSO

C NAME — FACE NAME

FmSetFunctionPrototype — **none**

SYNOPSIS

```
int     FmSetFunctionPrototype(
        String          Myfunction,
        String          MyCfunction,
        String          type1,
        String          type2,
...)
```

DESCRIPTION

Equivalent to the cprototype construct. The FmSetFunctionPrototype function takes a variable number of arguments. If the number of C types for the arguments of the FACE function does not match the number of arguments specified in the corresponding call to FmAttachFunction, the behavior is undefined.

Myfunction is the name of the function declared in a FACE script.

MyCfunction is the name the function should have in the C code.

type1 is the type of the return value in the C code.

type2 is the type of the first argument in the C code.

The types declared for *MyCfunction* (*type1*, *type2*, ...) should correspond to those declared for the FACE function MyFunction. If the number of arguments does not match, the behavior is undefined.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME | FACE NAME

FmSetSuperclassInfos | **none**

SYNOPSIS

```
int     FmSetSuperclassInfos(
        WidgetClass         *widget_class,
        char                *includes,
        char                *class_parts,
        char                *widget_parts,
        char                *class_init,
        char                *class_part_init,
        WidgetClass         *subclass_prototype,
        int                 subclass_prototype_size,
        void                (subclass_class_part_init)());
```

DESCRIPTION

Specify information on user-defined widgets. FmSetSuperclassInfos returns 1 on success, 0 on failure. It has the following parameters:

widget_class: here you should pass the same address as given in FmAddWidgetClass to identify the class.

includes specifies the include file(s) string to output in the C-code file containing the widget definition.

class_parts is a string containing the C declarations of the class parts of the class and its superclasses, up to and including Core.

widget_parts is a string containing the C declarations of the widget parts of the class and its superclasses, up to and including Core.

class_init is a string containing the C initializer of the widget class *not* including Core.

class_part_init is a string containing the C code to output in the ClassPartInitialize method or NULL.

subclass_prototype is a pointer to the data with which the class structure should be initialized when creating a sub--class of this class dynamically; the Core structure need not be initialized and can be all 0s.

subclass_prototype_size is the size of the data pointed to by *subclass_prototype*.

subclass_class_part_init() is a pointer to the method to be stored in the class part initialize of the class.

FmSetSuperclassInfos returns 1 on success, 0 on failure.

When using subclassed widgets created with XFaceMaker, the generated C-code file contains the function FmAdd*Name*WidgetClass which calls FmAddWidgetClass() with the appropriate values for the new widget class. The compilation flag XFM enables this code to link the new widget with XFaceMaker.

Note however it is up to you to register resource converters that may be required by your new widget, even if this is automatically done on widget instantiation, because XFaceMaker may have to parse the new resource types before this point.

The XFaceMaker Examples directory contains the sub--classed widget example TextValid.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmSetTallest **FmSetTallest**

SYNOPSIS

```
None        FmSetTallest(
            Widget      ...)
```

DESCRIPTION

FmSetTallest sets the height of up to twelve widgets to that of the tallest widget. If the list contains less than twelve widgets, you must put a null widget, Widget(NULL), after the last widget.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

FmSetWidest

C NAME FACE NAME

FmSetToplevel **none**

SYNOPSIS

```
void          FmSetToplevel(
              Widget        toplevel)
```

DESCRIPTION

Switch to a different display in multi–display applications. The current display becomes that of *toplevel*.

SCOPE

Application only

EXAMPLE

SEE ALSO

FmSetToplevelI

FmSetToplevelC

C NAME / FACE NAME

FmSetToplevelC **none**

SYNOPSIS

void FmSetToplevelC(
Widget *toplevel*)

DESCRIPTION

A specific version of FmSetToplevel defined in the Fm_c library.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

FmSetToplevelI **none**

SYNOPSIS

void FmSetToplevelI(
Widget *toplevel*)

DESCRIPTION

A specific version of FmSetToplevel defined in the Fm_e and Fm libraries.

SCOPE

Application only

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmSetValue **FmSetValue**

SYNOPSIS

```
Boolean     FmSetValue(
            Widget       widget,
            char         *name,
            XtPointer    address)
```

DESCRIPTION

This function calls the set script of the specified active value for the specified widget, passing it the specified address as the $ argument. This function is similar to FmSetActiveValue, but allows to specify the attached address for each call.

widget is the widget ID of the widget where the active value is defined.

name is the name of the active value

address is the address corresponding to the active value.

SCOPE

FACE scripts, application

EXAMPLE

SEE ALSO

FmAttachValue

FmGetValue

FmSetActiveValue

C NAME FACE NAME

FmSetWidest **FmSetWidest**

SYNOPSIS

```
None        FmSetWidest(
            Widget        ...)
```

DESCRIPTION

FmSetWidest sets the width of up to twelve widgets to that of the widest widget. If the list contains less than twelve widgets, you must put a null widget, Widget(NULL), after the last widget.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FmSetTallest

C NAME	**FACE NAME**
FmShowPopup	**ShowPopup**

SYNOPSIS

```
None        FmShowPopup(
            Pointer     event,
            Widget      menu)
```

DESCRIPTION

Pop up the *menu* popup-menu. The menu is popped so that its upper-left corner is at the position indicated by the *event*. This function is mostly used in a translation, where the event is passed as the $ special variable. XFaceMaker automatically defines such a translation on Button3 when a popup-menu is created.

SCOPE

FACE scripts   application

EXAMPLE

Translation defined in a bulletinBoard to open a popup menu (XFaceMaker defines a different translation):

```
translations = #override\
<ButtonPress>Button3:
eval("ShowPopup($,self.popup_PopupMenu0.PopupMenu0);”)
```

SEE ALSO

C NAME   FACE NAME

FmShowWidget   **Show**

SYNOPSIS

```
None        FmShowWidget(
            Widget      w)
```

DESCRIPTION

Display widget *w* on the screen. If *w* is a Shell widget, then Show realizes it and maps it. If *w* is a Popup-Shell, then Show calls XtPopup on the shell with a grab_kind of GrabNone. Otherwise, Show simply manages the Widget.

Note: if the intent is to show a DialogShell, *w* should refer to the DialogShell's child, not to the shell itself because of the way Motif handles DialogShells.

SCOPE

FACE scripts   application

EXAMPLE

Show a transientShell which is the shell ancestor of self:

Show(toplevel);

SEE ALSO

FmHideWidget

C NAME **FACE NAME**

FmStartCursorDrag **FmStartCursorDrag**

SYNOPSIS

```
Widget     FmStartCursorDrag(
           Widget      widget,
           XEvent      *event,
           String      targets,
           Pixmap      pixmap,
           Pixmap      mask)
```

DESCRIPTION

FmStartCursorDrag is like FmStartSimpleDrag, with the additional argument *mask* that specifies a mask bitmap to use with the cursor *pixmap*.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FmStartSimpleDrag

C NAME

FmStartSimpleDrag

FACE NAME

FmStartSimpleDrag

SYNOPSIS

```
Widget    FmStartSimpleDrag(
          Widget      w,
          XEvent      event,
          String      targets,
          Pixmap      pixmap)
```

DESCRIPTION

Start a drag operation.

w argument specifies the drag source. It is generally the widget in which the user has pressed mouse button two.

event is a pointer to the X event generated by the user's action (this must be a button event).

targets is a character string which contains the name(s) of the target(s) that the widget can provide. If several targets are specified, they must be separated by commas.

pixmap specifies the Pixmap which will be displayed during the drag operation by the drag icon. If *pixmap* is null, a default pixmap provided by Motif is used.

This function is generally called from a widget's translation table, in a FACE action script for a button press. In this case, the widget argument is *self* and the event argument is *$*. Similar to FmDragStart, but the argument list is replaced by a single *pixmap* argument. This pixmap is used to create a drag icon using XmCreateDragIcon. The icon is then passed as the XmNsourcePixmapIcon argument to XmDragStart.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

FmDragStart

C NAME FACE NAME

FmStopTimeOut **StopTimeOut**

SYNOPSIS

None FmStopTimeOut(
TimeOutClosure *TimeoutID*)

DESCRIPTION

Stop a previously started timeout identified by the TimeoutID returned by FmCallCallbacksTimeOut, FmSetActiveValueTimeOut, or FmGetActiveValueTimeOut.

Before calling this function, you must make sure that the time out has not already been triggered.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

FmCallCallbacksTimeOut

FmGetActiveValueTimeOut

FmSetActiveValueTimeOut

C NAME — FACE NAME

FmWait — **wait**

SYNOPSIS

```
None        FmWait(
            Widget      w,
            Int         delay)
```

DESCRIPTION

Wait during *delay* milliseconds, handling X events meanwhile.

SCOPE

FACE scripts

EXAMPLE

Wait one second:

```
wait(self,1000);
```

SEE ALSO

C NAME　　　　　　　　　　　　FACE NAME

FmWaitForReturn　　　　　　　　**WaitForReturn**

SYNOPSIS

String FmWaitForReturn()

DESCRIPTION

Enter a local event-handling loop, until Return is called. Return the value passed to Return as an argument. For example:

```
Popup(root.PopupShell, 2); result = WaitForReturn();
```

is equivalent to:

```
result=PopupAndWait(root.PopupShell);
```

SCOPE

FACE scripts　　application

EXAMPLE

A push button shows a transientShell and waits for a return value; the transientShell contains a messageBox whose okcallback calls Return: in the push-button:

```
activateCallback = Show(root.TransientShell0);
str=WaitForReturn();
printf("%s\n”,str);
```

in the MessageBox:

```
okCallback = Return(self,"my returned string"); Hide(toplevel);
```

This is equivalent to: in the pushButton:

```
activateCallback = str = PopupAndWait(root.TransientShell0);
printf("%s\n”,str);
```

in the MessageBox:

```
okCallback = Return(self,"my returned string");
Hide(toplevel);
```

SEE ALSO

FmPopupAndWait, FmReturn

C NAME

FmWarpPointer

FACE NAME

WarpPointer

SYNOPSIS

```
None        FmWarpPointer(
            Widget      w)
```

DESCRIPTION

Move the mouse pointer to the center of widget *w*. Should be used in rare cases: taking control of the pointer independently from the user is contrary to good ergonomics and to Motif Style guide recommendations.

SCOPE

FACE scripts	application

EXAMPLE

In a pushButton, force the pointer to move to the center of a sibling drawingArea:

```
WarpPointer(parent.XmDrawingArea0);
```

SEE ALSO

C NAME / FACE NAME

FmWidest | **FmWidest**

SYNOPSIS

```
Int         FmWidest(
            Widget        ...)
```

DESCRIPTION

FmWidest returns the width of the widest widget of up to twelve widgets, the maximum number of arguments in a FACE function call. If the list contains less than twelve widgets, you must put a null widget, Widget(NULL), after the last widget.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

C NAME FACE NAME

FmWidgetName **WidgetName**

SYNOPSIS

```
String      FmWidgetName(
            Widget      widget)
```

DESCRIPTION

Return the name of widget *widget*. The string is returned in a static buffer and should not be freed.

SCOPE

FACE scripts, application

EXAMPLE

SEE ALSO

C NAME

FmWriteValue

FACE NAME

none

SYNOPSIS

```
int         FmWriteValue (
            Widget      widget,
            char        *av_name,
            XtArgVal    value,
            String      type)
```

DESCRIPTION

This function fetches the address attached to the specified active value *av_name* and associated with the specified widget *widget* (possibly allocating a memory location of size sizeof(XtArgVal) if the active value is automatic), converts *value* to the type of the active value if *value* is of type String, stores *value* as the contents of the active value address, and calls the set script of the active value with the address passed as the $ argument. If it succeeds, the function returns 1. If the active value does not exist, or if it is not attached and it is not automatic, the function returns 0. If it is immediate, or if *widget* is null, or if the value cannot be converted to the active value type, the function returns -1.

Calling this function is similar to executing an instruction of the form:

<widget>:@<name> = value;

in a FACE script.

SCOPE

Application

EXAMPLE

SEE ALSO

C NAME | FACE NAME

SuperclassDeleteChild | **SuperclassDeleteChild**

SYNOPSIS

```
SuperclassDeleteChild(
Widget      parent,
Widget      w)
```

DESCRIPTION

Call the **DeleteChild** method of the superclass of widget *parent*, to delete *w* from the list of children.

In general, if you have defined a **DeleteChild** method when defining a new class, you must call this function to delete a widget *w* from its parent's children array. Otherwise, widget *w* would remain managed by its parent, and this could cause a crash.

SCOPE

FACE scripts, application

EXAMPLE

```
SuperclassDeleteChild(self, $);
```

SEE ALSO

SuperclassInsertChild

C NAME — FACE NAME

SuperclassExpose — **SuperclassExpose**

SYNOPSIS

```
None        SuperclassExpose(
            Widget      w,
            Xevent      *event,
            Region      region)
```

DESCRIPTION

Call the expose method of the superclass of widget *w*, if *w* is realized.

In general, you should call this function after the class Expose method, if you have defined such a method when defining a new class.

SCOPE

FACE scripts application

EXAMPLE

SuperclassExpose(self, @, @1);

SEE ALSO

C NAME FACE NAME

SuperclassInsertChild **SuperclassInsertChild**

SYNOPSIS

```
None        SuperclassInsertChild(
            Widget      parent,
            Widget      w)
```

DESCRIPTION

Call the **InsertChild** method of the superclass of widget *parent*, to insert *w* in the list of children.

In general, if you have defined an **InsertChild** method when defining a new class, you must call this function to insert a new widget *w* in its parent's children array. Otherwise, widget *w* would not be managed by its parent.

SCOPE

FACE scripts, the application

EXAMPLE

```
SuperclassInsertchild(self, $);
```

SEE ALSO

C NAME FACE NAME

SuperclassResize **SuperclassResize**

SYNOPSIS

```
None        SuperclassResize(
            Widget      w)
```

DESCRIPTION

Call the resize method of the superclass of widget *w*, if *w* is realized.

In general, you should call this function after the class Resize method, if you have defined such a method when defining a new class.

SCOPE

FACE scripts, the application

EXAMPLE

SEE ALSO

C NAME FACE NAME

atoi **atoi**

SYNOPSIS

```
Int        atoi(
           String      s)
```

DESCRIPTION

Call the C function atoi.

SCOPE

FACE scripts, the application

EXAMPLE

SEE ALSO

C NAME / FACE NAME

C NAME	FACE NAME
none	**breakpoint**

SYNOPSIS

```
Boolean     breakpoint(
            Widget      widget
            String      message)
```

DESCRIPTION

Explicitly call a breakpoint within a FACE script. The widget and the message are displayed when the breakpoint is activated.

SCOPE

FACE scripts

EXAMPLE

SEE ALSO

C NAME

none

FACE NAME

cbtoi

SYNOPSIS

```
Int cbtoi (
        Int     value)
```

DESCRIPTION

Convert a byte value to an integer. Although FACE always uses XtArgVal-sized integers, cbtoi returns an integer within the appropriate range for a byte.

SCOPE

FACE scripts

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

none **cftoi**

SYNOPSIS

```
Int cftoi (
        Float       f)
```

DESCRIPTION

Convert a floating-point number to an integer.

SCOPE

FACE scripts

EXAMPLE

Perform a calculation in Float and convert to Int.

```
Float ymin;
Float ymax; Float y;
Int v;
min = self:sliderMin;
max = self:sliderMax;
v = cftoi((y - ymin) * citof(max - min) / (ymax - ymin));
```

SEE ALSO

citof

C NAME **FACE NAME**

none **citof**

SYNOPSIS

```
Float citof (
            Int      i)
```

DESCRIPTION

Convert an integer to a floating-point number.

SCOPE

FACE scripts

EXAMPLE

Perform a calculation in Float and convert to Int.

```
Float ymin;
Float ymax; Float y;
Int v;
min = self:sliderMin;
max = self:sliderMax;
v = cftoi((y - ymin) * citof(max - min) / (ymax - ymin));
```

SEE ALSO

cftoi

C NAME **FACE NAME**

none **clear**

SYNOPSIS

```
None clear (
        FaceArray      a)
```

DESCRIPTION

Delete all entries in array *a*, but not the array itself.

SCOPE

FACE scripts

EXAMPLE

Clear a FACE array that has been declared elsewhere:

```
global FaceArray a;
clear(a);
```

SEE ALSO

destroy

C NAME | FACE NAME

none **cstoi**

SYNOPSIS

```
Int cstoi (
          Int       value)
```

DESCRIPTION

Convert a short value to an integer. Although FACE always uses XtArgVal-sized integers, cstoi returns an integer within the appropriate range for a short.

SCOPE

FACE scripts

EXAMPLE

SEE ALSO

C NAME **none** / FACE NAME **data**

SYNOPSIS

```
Pointer data(
        FaceArray    a)
```

DESCRIPTION

Return a pointer to the data in FaceArray *a*.

SCOPE

FACE scripts

EXAMPLE

Get the address of a FACE array that has been declared elsewhere:

```
global FaceArray a;
p = data(a);
```

SEE ALSO

new_c_array

new_array

C NAME

none

FACE NAME

destroy

SYNOPSIS

```
None destroy (
          FaceArray      a)
```

DESCRIPTION

Free array a.

SCOPE

FACE scripts

EXAMPLE

Free a FACE array that has been declared elsewhere:

```
global FaceArray a;
destroy(a);
```

SEE ALSO

clear

C NAME **FACE NAME**

exit **Quit**

SYNOPSIS

```
None        exit(
            Int          exitVal)
```

DESCRIPTION

Quit the application or the interface. If called in Try mode, will only print a message in the Messages Box. The argument *exitVal* will be returned to the calling program, normally a Unix shell. It should follow the Unix conventions.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

none **first**

SYNOPSIS

```
key_type first (
                    FaceArray     a,
                    key_type      k)
```

DESCRIPTION

Return the initial key for array *a. k* is the key value returned if the array is empty.

SCOPE

FACE scripts

EXAMPLE

```
Int b[String] = new_table();
key1 = "key1";
b[key1] = 1;
key2 = "key2";
b[key2] = 2;
String si = first(b, 0);
printf("  key first = %s\n", si);
printf("first = %d\n", b[si]);
String s2 = next(b,si,  0);
printf("  key second = %s\n", s2);
printf("second = %d\n", b[s2]);
```

SEE ALSO

next

size

C NAME **FACE NAME**

none **new_array**

SYNOPSIS

FaceArray new_array ()

DESCRIPTION

Create a new linear array.

SCOPE

FACE scripts

EXAMPLE

Declare, initialize, and print the content of a FACE array:

```
String a[Int] = new_array();
a[0] = "zero";
a[1] = "one";
a[2] = "two";
a[3] = "three";
for (i = 0; i < size(a); i=i+1)
printf("index: %d val: %s \n",i,a[i]);
```

SEE ALSO

new_c_array

new_table

C NAME **FACE NAME**

none **new_c_array**

SYNOPSIS

```
FaceArray new_c_array (
                Pointer     pa,
                Int         i)
```

DESCRIPTION

Create a new linear array, allowing you to initialize the array with a C array. *pa* is a pointer to the C array, and *i* is the size of the C array.

SCOPE

FACE scripts

EXAMPLE

Declare and initialize a FACE array to contain the items of an XmList

widget; print its contents:

```
XmString a[Int];
count = parent.XmList0:itemCount;
a = new_c_array(Pointer(parent.XmList0:items),count);
for (i=0; i < count; i=i+1)
printf("index: %d item %s\n",i,GetString(a[i]));
```

SEE ALSO

new_array

new_table

C NAME / FACE NAME

none **new_struct**

SYNOPSIS

```
MyStruct new_struct (
                MyStruct      ms)
```

DESCRIPTION

Allocate a new structure in a FACE script. *ms* is the name of the structure of type MyStruct. When the structure is no longer needed, use free or XtFree.

SCOPE

FACE scripts

EXAMPLE

Declare a structure which contains a FACE array, and initialize the array:

```
struct  Mystruct {
        Int i;
        FaceArray a;
        Int j;
};
global Mystruct s;
s = new_struct(Mystruct);
s->a = new_array();
String a[Int] = s->a;
a[0] = "zero";
a[1] = "one";
a[2] = "two";
a[3] = "three";
```

SEE ALSO

FmRegisterStructures

C NAME **FACE NAME**

none **new_table**

SYNOPSIS

FaceArray new_table ()

DESCRIPTION

Create a new hash array.

SCOPE

FACE scripts

EXAMPLE

```
Int b[String] = new_table();
key1 = "key1";
b[key1] = 1;
key2 = "key2";
b[key2] = 2;
String si = first(b, 0);
printf("  key first = %s\n", si);
printf("first = %d\n", b[si]);
String s2 = next(b,si,  0);
printf("  key second = %s\n", s2);
printf("second = %d\n", b[s2]);
```

SEE ALSO

new_array

new_c_array

C NAME — FACE NAME

none — **next**

SYNOPSIS

```
key_type next (
                FaceArray    a,
                key_type     p,
                key_type     k)
```

DESCRIPTION

Return the next key in array *a*. *p* is the previous key returned by first or next, and *k* is the key to be returned if the end of the array is reached.

SCOPE

FACE scripts

EXAMPLE

```
Int b[String] = new_table();
key1 = "key1";
b[key1] = 1;
key2 = "key2";
b[key2] = 2;
String si = first(b, 0);
printf("  key first = %s\n", si);
printf("first = %d\n", b[si]);
String s2 = next(b,si,  0);
printf("  key second = %s\n", s2);
printf("second = %d\n", b[s2]);
```

SEE ALSO

first

size

C NAME	**FACE NAME**
printf	**printf**

SYNOPSIS

```
None        printf(
            None   f
            ...)
```

DESCRIPTION

Call the C function printf. No type checking is done.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

none **random**

SYNOPSIS

```
Int random (
        Int     num)
```

DESCRIPTION

Return a random integer whose value lies between 0 and num.

SCOPE

FACE scripts only

EXAMPLE

In a pushbutton, set the labelString to a random value between 0 and 300, as long as the button is armed. Get a new random number every half second.

```
armCallback = global t;
global i;
t = init_timer(self, "global i; i=random(300);
        self:labelString=itoa(i);",0,500,True); disarmCallback = global t;
stop_timer(t);
```

SEE ALSO

C NAME   FACE NAME

none   **size**

SYNOPSIS

```
Int size (
        FaceArray      a)
```

DESCRIPTION

Return the number of elements in array *a*. For hash arrays, it is the number of values stored in the array. For linear arrays, it is the largest index used to store a value minus one, or the number of elements specified for an initialized array.

SCOPE

FACE scripts

EXAMPLE

SEE ALSO

C NAME   FACE NAME

sprintf **sprintf**

SYNOPSIS

```
None sprintf(
        None        targ,
        None        f,
        ...)
```

DESCRIPTION

Call the C function sprintf. No type checking is done.

SCOPE

FACE scripts   application

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

sscanf **sscanf**

SYNOPSIS

```
None        sscanf(
            None        src,
            None        format,
            ...)
```

DESCRIPTION

Call the C function sscanf. No type checking is done.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

C NAME **FACE NAME**

strcat **strcat**

SYNOPSIS

```
None        strcat(
                String      s1,
                String      s2)
```

DESCRIPTION

Call the C function strcat.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

C NAME FACE NAME

strcmp **strcmp**

SYNOPSIS

```
Int     strcmp(
        String      s1,
        string      s2)
```

DESCRIPTION

Call the C function strcmp.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

C NAME FACE NAME

strcpy **strcpy**

SYNOPSIS

```
None strcpy(
        String      targ,
        String      src)
```

DESCRIPTION

Call the C function strcpy.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

C NAME | FACE NAME

strlen **strlen**

SYNOPSIS

```
Int     strlen(
        String          src)
```

DESCRIPTION

Call the C function strlen.

SCOPE

FACE scripts application

EXAMPLE

SEE ALSO

C NAME	**FACE NAME**
strncmp	**strncmp**

SYNOPSIS

```
Int strncmp (
        String      s1,
        String      s2,
        Int         i)
```

DESCRIPTION

Call the C function strncmp.

SCOPE

FACE scripts	application

EXAMPLE

SEE ALSO

CHAPTER 13: Command Line Options

This chapter lists the command line options available for use in XFaceMaker. Each option has an associated resource whose name is listed in bold.

13.1 The Command Line Options

The full command line is:

```
xfm [options] <file>.fm
```

or if you are loading XFM *and* the Draw Widget Library:

```
xfmwf [options] <file>.fm
```

13.1.1 General Options

-help : (**help**) Print the command line information given here.

-lazy : (**lazy**) Specify that XFaceMaker interface widgets are created when needed.

-nonlazy : (**lazy**)This is the opposite of -lazy, which is the default in this release. If you specify -nonlazy, all the windows of XFaceMaker will be created at startup time, otherwise, they will be created the first time they are used.

-postordermanage : (**postOrderManage**) In interpreted mode, specify that children are managed first (compatibility with v2.1).

-revision: (**revision**) Print revision number for this release of XFaceMaker.

13.1.2 Environment Options

-appclass <application shell name>:(**applicationClassName**) Specify XFaceMaker's default application shell class name. Defaults to XfmApplication if not specified.

-autoplace <true/false>:(**autoPlace**) Specify automatic/interactive placement of XFM windows.

-bufsize <number>: (**bufferSize**) Specify size of internal buffer used to display resources, scripts, etc.

-calleditor : (**callEditor**) Call the FmCallEditor function when entering FmLoop.

-classes <classes>: (**startupClasses**) Specify classes to load at startup as a comma-separated list of classes. The syntax is

Class1:Classfile1.fm,Class2:Classfile2.fm

If the name of the .fm file is the same as that of the class, you can specify Class only. For example, to load classes C11 and C12, you could specify C11:C11.fm,C12:C12.fm or just C11,C12.

-coloricons <true/false>: (**colorIcons**) On color screens, specify whether to use/not use color icons in the XFaceMaker windows.

-defaultnames <names> : (**defaultNames**) Specify default object names template. Specify a comma separated list of default object names; e.g. XmBulletin-Board=BB,XmPushButton=PB,...

-display <displayname:0.0> : (**display**) Standard X client option, specifies the display to be used. May be used instead of defining the DISPLAY environment variable.

-fastlist : (**fastResourceList**) This option specifies that the "fast" resource list be displayed in the Resources box instead of the list containing icons, which is the default. The "fast" list option refreshes resources faster on edit widget change.

-fmbootfile <alternate boot file> : (**fmbootfile**) The default boot file is Fm.fm. You should not need to change it. This specifies which .fm file is to be loaded and executed by XFaceMaker.

-fmcolors <name> : (**fmcolors**) Specify <name> as the .rdb file for XFaceMaker interface colors.

-fmfonts <name> : (**fmfonts**) Specify the name of the resource file to be used. By default, the file XfmDb in the directory $NSLHOME/lib/xfm or the directory specified by -fmlibdir is used.

-fmlibdir <alternate_fm_directory> : (**fmlibdir**) Specify an alternate directory from which to fetch the XFaceMaker interface description file, Fm.fm. The default directory is /usr/lib/X11/xfm for installations in the default directory, $NSL-HOME/lib/xfm for installations in a user-specified directory (where **NSLHOME** is set to the user-specified directory), /usr/local/lib/xfm for installations in /usr/local. The environment variable **FMLIBDIR** may be used instead of the command line option.

-fmresources : (**fmfonts**) This option has been kept for backward compatibility only. Use -fmfonts<name> instead.

-ignorexerrors : (**ignorexerrors**) Ignore "Bad Drawable" X11 protocol errors generated by some versions of the OSF/Motif Toolkit.

-nocalleditor : (**callEditor**) Do not call the FmCallEditor function when entering FmLoop.

-token <type> : (**token**) Specify which token type you wish to use. When different versions of XFaceMaker are available from the same token server a particular token can be requested. Current types are XFM, XFM Entry, and XFM IDT. When using NSLd license server, token names are: 007, 008, 006 respectively. When using the NSLLicense license server, token names are: xfm3, xfm3/el, xfm3/idt, xfm3/demo.

13.1.3 Editor Options

-build <char> : (**build**) Specify <char> as the accelerator to switch to *Build* mode.

-copy <char> : (**copy**) Specify <char> as the accelerator for the *copy widget* operation.

-cut <char> : (**cut**) Specify <char> as the accelerator for the *cut* operation.

-duplicate <char> : (**duplicate**) Specify <char> as the accelerator for the *duplicate widget* operation.

-hidetmplres : (**hidetmplres**) This option causes resources inherited from templates to be omitted from the Resources list.

-kill <char> : (**kill**) Specify <char> as the accelerator for the *delete widget* operation.

-nonuniquenames : (**nonuniquenames**) This disables the XFaceMaker feature that forces widget names to be unique by adding a suffix, allowing different children of the same parent widget to have the same name. This lets resource specification files use common widget names.

-paste <char> : (**paste**) Specify <char> as the accelerator for the *paste widget* operation.

-try <char> : (**try**) Specify <char> as the accelerator to switch to *Try* mode.

-undo <char> : (**undo**) Specify <char> as the accelerator for the *undo* operation.

13.1.4 Compilation Options

When using the compilation commands the interface of XFaceMaker is not loaded, and it exits immediately after the compilation without displaying windows on the screen. However, XFaceMaker still connects to the X Display, so the X server must be running and the DISPLAY environment variable must be set correctly. This allows certain X data structures and functions to be used to validate the input file.

-ansic : (**ansic**) Generate C-code with prototyped functions conforming to the *ANSI* C standard. This is the default.

-c++ : (**cxx**) Generate a C++ class.

-c++lib < name> : (**cxxClassLib**) Specify the base C++ class library.

-c++name <name> : (**cxxClassNam**) Specify the C++ class name.

-c++file <name>: (**cxxCFile**) Specify the C++ implementation file name.

-c++header <name> : (**cxxHFile**) Specify the C++ header file name.

-cfile <file.c> : (**cfile**) Explicitly specify the name for the C file generated by the -compile or -compilegroup options. The default is to use the .fm basename with the suffix .c.

-cflags <character> : (**cflags**) Specify an option for C-code generated by the -compile or -compilegroup options with a character. Characters may be concatenated to give a string of options. See the Command Line Options table for character values.

-cinclude <string> : (**cinclude**) Specify an include file name for the C file generated by the -compile or -compilegroup options. The string is inserted directly after the text #include. Quotes and other characters must be protected when defined from the shell. e.g. xfm -compile my.fm -cinclude’"myinclude.h"’

-cname <name> : (**cname**) Specify a different name for the main function FmCreate<name> in the C file generated by the -compile or -compilegroup options. The default is to use the name FmCreate<filename> where filename is the basename of the .fm file being compiled.

-compile : (**compile**) Compile a .fm file to create a C file. If the option -rdb is used, then <file>.rdb is created together with the C-code version of <file>.fm_rdb. All references to unknown widgets are resolved dynamically.

-compilegroup : (**compilegroup**) Compile a .fm group file to create a C-code file. All references to widgets are resolved dynamically.

-coptions <number> : (**coptions**) Obsolete, use cflags instead.

-cprefix <Prefix> : (**cprefix**) When generating a widget class this option defines the prefix string to be used for the widget class as in Xm or Xnsl.The prefix defined is used in much the same manner as that of Xm for the OSF/Motif widgets. The class name becomes Prefix*Name*, the include files are expected to be in the Prefix directory, (as in #include<Prefix/Name.h>), and resource values begin with the prefix.

-force: With this option set, XFaceMaker stays silent when genappli finds application or interface C code files whose timestamp has changed. If this option is not set, genappli pops a confirmation dialog when it encounters files whose timestamp has changed. This may

-instclose : (**instclose**) Used with the -compile or -compilegroup options, causes XFaceMaker to install, on shells, a default handler that will be called when the Motif Window Manager mwm *Close* menu item is selected, as in interpreted Fm mode. The default handler exits the application if the shell is an ApplicationShell, or unmaps the shell otherwise. To change the handler use FmSetCloseHandler.

-msg : (**msg**) Assign all message numbers in <file>.fm, and generate a message catalog source file corresponding to all the internationalized strings in <file>.msg. <file>.fm is changed if some message numbers were not assigned previously, i.e., all messages are numbered.

-nowarnings : (**nowarnings**) This option prevents the FACE compiler from issuing certain frequent warnings when compiling FACE scripts into standalone C code, i.e. when the -standc option is used. The deleted warnings are: Resource <resource name> might need conversion and Unknown resource <resource name>. Note that these warnings should be suppressed or ignored only if you are sure that the specified resource does not need special conversions. See also -silentmode.

-rdb : (**rdb**) Create files <file>.fm_rdb and <file>.rdb. In conjunction with -compile or -compilegroup the C-code version of the file <file>.fm_rdb is also generated.

-rdbclass <name>: (**rdbclass**) When used in conjunction with -rdb, this option specifies the first component of the resource specification to be *name*.

-rdbresources <type,...> : (**rdbresources**) Specify resource types or names to be saved in RDB resource files

-standc : (**standc**) Used with the -compile option causes the generation of *standalone* C-code; i.e., C-code that need not be linked with the Fm_c library.

-startup <file.fm> : (**startup**) This option allows an FACE script file to be loaded before compilation begins. Ordinarily, this is used to load the description of a widget sub–class which exists in the .fm file to be compiled but has not been linked into XFaceMaker.

-silentmode : (**silentmode**) This option disables all warning messages from the FACE interpreter and C-code generator. This can be used to suppress warning messages issued by FACE when generating standalone C code, if you are sure that all resource conversions in your scripts are OK. See also nowarnings.

-uil : (**uil**) Compile a .fm file to create a UIL file. If a C-code file is generated at the same time, all widget references are resolved dynamically. The Cfile generated contains all the scripts and none of the interface description which is in the UIL file.

-uildialogs : (**uildialogs**) Causes XFaceMaker to generate predefined combinations of objects called “Dialogs” in the UIL file. Can only be used in conjunction with the -uil option..

	Option	Cmd Line Option	-cflags	Use with
1	Creation function	-compile		
2	Generate widget class		g	1
3	Class prefix	-cprefix <Prefix>		2
4	Creation name	-cname <Name>		1
5	Include file	-cinclude <Text>		1
6	K &R C		k	1
7	ANSI C	default	C	1
8	Use resource database		r	1
9	Use XtSetArg for all resources	default	c	1
10	Use XtVaCreateWidget		v	1
11	Convert resources after creation		s	9 Only
12	Convert Pixmaps after creation		p	9 Only
13	Create XmStrings		x	9 Only
14	5 creation arguments allowed	default		10 Only
15	20 creation arguments allowed		m	10 Only
16	100 creation arguments allowed		M	10 Only
17	Store Widgets in static variables		l	1
18	Store Widgets in allocated array	default	a	1
19	Store Widgets in extern variables		G	
20	Use Fm_c library	default		1
21	Standalone C	-standc	S	1
22	Install WM `Close' Handler	-instclose	i	1
23	Dynamic Widget references	-compilegroup	d	Instead of 1

Table 13.1: C-Code Command Line Options

13.1.5 Dual Process Options

-autoclient : (**autoClient**) Launch client automatically in server mode.

-autosave <number> : (**autoSaveInterval**) Specify interval (in seconds) of automatic saving of design process state.

-client : (**client**) Run in client mode

-designprocess <name>: (**designProcess**) Specify the design process to be launched by XFaceMaker at startup.

-monoprocess : (**monoprocess**) Run XFaceMaker in mono-process mode.

-nointerface : (**nointerface**) Do not load XFaceMaker's interface.

-projmgrname <name> : (**projectManagerName**) Specify the name used by the project manager process to connect to XFaceMaker.

-restore : (**restore**) Restore editing state stored in the XFaceMaker server.

-server : (server) Run in server mode.

-serverhost <name> : (**serverHost**) Specify the name of the host where the editing server runs.

-servername <name>: (**serverName**) Specify the name with which the XFaceMaker server is registered.

-socketfile <name> : (**socketFile**) Specify an alternate UNIX domain socket path name.

-socketport <number> : (**socketPort**) Specify an alternate INTERNET domain socket port number.

-toolkit <name> : (**toolkitClasses**) Specify a comma separated list as the initial toolkit; e.g. -toolkit Primitive,Shell,Composite

-trusted : (**trusted**) Restore trusted editing state stored in XFaceMaker server.

13.1.6 Application Generation Options

-applifiles <name> : (**appliFiles**) List of application files to generate.

-cleanappli : (**cleanAppliFiles**) Generate “clean” application files (without code preservation information).

-cpattfiles <files>: (**cPattFiles**) This option is reserved for future use.

-genappli : (**generateAppliFiles**) Generate application files.

-onlygen : (**onlyGen**) Generate application file template only (use with -genappli).

-preserve : (**cleanAppliFiles**) Generate application files with code preservation information.

-target <Target>: (**targetEnvironment**) This option is used to specify the environment for whichthe interface and the application files are to be generated, e.g. Windows.

13.1.7 Debug Options

-debugface : (**debugFace**) Enable debugging of FACE scripts.

-nodebugface : (**debugFace**) disable debugging of FACE scripts.

-dumpcore : (**dumpcore**) Cause a core dump on receipt of signal 11. By default XFaceMaker will exit without producing the file core on receipt of signal 11.

-synchronous : (**synchronous**) Standard X client debug flag.

-verbose : (**verbose**) Print messages during execution.

13.1.8 XFaceMaker Extension Options

-extensions "*<ext1> <ext2> ...*": Specifies FACE extension description files to load at startup.

-extlib *<library-name>*: Used with the -gen option to generate the source files of an extension library. The pattern files for extension libraries are PattLibMake and PattLibExt.c.

- newxfm *<xfm-name>*: Used in conjunction with the -gen option to generate the source files of a new XFaceMaker. The pattern files used are PattXfmMake and PattXfmMain.c.

-extpath *<dir1/%F:dir2/%F...>*: Search path for extension files.

13.1.9 Standard X client

All standard X client command line options will be recognized although not all may be meaningful for XFaceMaker . For a full list consult the standard X documention. The *X Window System User's Guide* from O'Reilly has a description in the chapter “Command Line Options".

Finally <file>.fm:

file The name of the interface file that you wish to load with XFaceMaker if you want to load it directly and not with the *Open* command in the File menu.

All of the above command line options can be set in your $HOME/.Xdefaults file, although only a few are really meaningful. This is done by using the option name without -, followed by a colon, a tab character, and the value required.

For example, to disable lazy mode every time you use XFaceMaker add the following to your $HOME/.Xdefaults file at the beginning of a new line.

```
Xfm.nonlazy:	true
```

Other examples are:

```
Xfm.fmresources: /home/my/xfm/MyXfmDb
Xfm.kill:	\^X
```

When using the CallEditor function any command line options for the application will be used by XFaceMaker when the function is called.

CHAPTER 14: Environment Variables

The following environment variables may be used with XFaceMaker or its applications. You can set them from within XFaceMaker, through the

Windows menu in XFaceMaker's main window.

DISPLAY Standard X display variable. The option -display may also be used.

FMEDITOR Defines the text editor that XFaceMaker will call when the Custom Editor button is selected in one of the XFaceMaker editing windows. Set this to *wx242* to use NSL's text editor. You may set it to *vi* or *emacs* if you prefer. Make sure they are included in your PATH.

FMFILEPATH An environmental variable used by XFaceMaker to define the directories searched to find all Fm files. %F is used as the substitution character for the .fm file name. For example, if your .fm files are in the directory /dir/Appli/Fm, set this variable as follows:

/dir/Appli/Fm/%F

FMLIBDIR An environment variable which may be used to specify the directory where the XFaceMaker interface description file Fm.fm should be fetched. See also the -fmlibdir command option.

FMPIXMAPSPATH An environment variable used by XFaceMaker to define the directories searched to find color pixmap files. %P is used as the substitution character for the XWD file name.

LANG Defines the language element of NLSPATH.

LD_LIBRARY_PATH (sun,dec,sgi,sco)

LIBPATH (ibm)

LPATH (hp) Standard environment variable to specify search paths for libraries when loading a dynamically linked executable. The name differs on different operating systems. When setting this variable, include the directory $NSLHOME/lib .

NLSPATH When using internationalized interfaces this defines the directories searched to find the message catalog files.

NSL_LICENSE_HOST Specifies the name of the host machine for the server if you are using NSLlicense.

NSL_LICENSE_PORT specifies the port number the product should use to communicate with the license server. The default is 5002.

NSLHOME Used to access the XFaceMaker auxiliary files such as include files, application patterns, libraries, XFaceMaker interface files, etc. The default value of FMLIBDIR is initialized to $NSLHOME/lib/xfm.

PATH Standard environment variable to specify directories where executables should be fetched. When setting this variable, include the directory $NSLHOME/bin .

XAPPLRESDIR Standard X variable used to define the directories searched to find application resource files.

XBMLANGPATH Standard X variable used to define the directories searched to find bitmap files. %B is used as the substitution character for the bitmap file name.

XFM_LICENSE_NAME Specifies the name of the license if you are using NSLlicense as the license server daemon. This should be set to xfm3 or xfm3/el, or xfm3/idt depending on the type of product you have purchased.

Obsolete - XFaceMaker version 3.2.2 and above is incompatible with NSLd:

NSLSERVER If you are using the NSLd token server, this environment variable is used to specify the name of the machine running the NSL token server, from which tokens will be fetched. Useful especially when several NSL token servers are running on a network.

CHAPTER 15: The Fm File Structure

This chapter describes the Fm file, to enable developers to read or edit a file. An Fm file consists of a number of different blocks:

widget Each widget in an Fm file starts with:

```
widget <Widget Class>
```

followed by an opening brace ({) denoting the start of the widget block. The block continues until the matching close brace (}).

flag The flag block follows the widget block. It can contain the following fields:

- **mapped** Present if the widget was mapped when the Fm file was saved. If a parent of the widget was not mapped, the widget may not be visible.
- **popup** Present if the popup togglebutton in the Resources window was set. The widget is created as a PopupShell.
- **autosize** Present if the autosize toggle button in the Resources window was set. This denotes that the width and height resources will be ignored when creating the widget.
- **autopos** Present if the autopos toggle button in the Resources window was set. This denotes that the x and y resources will be ignored when creating the widget.
- **gadget** Present if the gadget toggle button in the Resources window was set. This denotes that the gadget version of the widget will be created.
- **menu_bar** Present for XmRowColumn widgets if they are acting as a menu bar.
- **menu_pulldown** Present for XmRowColumn widgets if they are acting as a pulldown menu.

active value	The flag block may be followed by one or more active value blocks which define any active values associated with the widget, along with their Get and Set scripts.
createproc	The flag block may be followed by a createproc block which defines the xfmCreateCallback script for this widget.

template	If an interface contains one or more templates then a template block defines the template files to be included.
resources	The resources block contains all resources defined for this widget, including callbacks and translations. The resources defined may be modified by the flag block entries.
children	If the widget is a parent, then a children identifier will mark the start of the child widget definitions block. Further widget definition blocks will occur within this block.

Finally, please note the following points:

- The name of the widget comes after the closing brace of the widget block.
- A line starting with the characters # or ! is a comment and therefore ignored.
- The indentation of a widget block in the Fm file shows the relationships of the widget, i.e., a child is indented with respect to its parent. When reading a file this indentation is *not* taken into account to determine the relationship between widgets, this depends entirely on the children block marker. Indentation uses the space character; tab characters exist only as formatting characters in FACE scripts where it represents the continuation of the previous line at the start of a line. Beware of editors which insert tab characters automatically after newlines.
- The commands xfmprint, xfmtableprt and fmavprt can give you a compact view of the interface if they are available on your system. They are described in the appendix.

CHAPTER 16: Adding Help Files

The XFaceMaker library Help directory contains ASCII files which provide the text for the Help menu. You may change the text as appropriate to local conditions or add your own windows. The files use the following rules:

- Each new item begins on a new line with the item name.
- Item names must not have leading or following space characters.
- Every text line for an item begins with <tab>.
- Every *blank* line required must begin with a <tab>.

To include an extra help window, add an XmPushButton to the Help Menu of Fm.fm with the following callback:

```
activateCallback = \
Help("BoxTitle", FmGetHelp("FileName", "item"));
```

The best example to follow is the *On Mouse* PushButton of Fm.fm.

CHAPTER 17: Porting the XFaceMaker C-Code Library

This chapter gives instructions to recompile the XFaceMaker run-time support library, libFm_c to a target platform on which the library is not already available.

17.1 Source and Configuration Files

Once XFaceMaker is installed, the files needed to recompile the libFm_c library are in the directories $NSLHOME/src/Clib and $NSLHOME/src/Clib/config.

17.2 Generating the Makefile and Compiling

17.2.1 Platforms supported by NSL

If the platform is one of the reference platforms (Sparc SunOS 4.1.x, Sparc Solaris 2.5, IBM RS/6000 AIX 4.1.5, DEC Alpha OSF/1 v3.2 and up, SGI IRIX 5.3, SCO Open Server 5 (SCO_SV sco 3.2 2), or HP HP-UX 9.0), the existing configuration files should work. All you have to do is follow the steps below:

1. cd $NSLHOME/src/Clib

2. config/boot

The boot script asks you a few questions: type Return each time. It builds the imake file

3. config/imake -Iconfig

4. make

Wait while the source files are being compiled.

17.2.2 Platforms not supported by NSL

If you want to compile the libFm_c library for another platform, you will have to modify the following files in the config directory first:

- Imake.tmpl (template file)
- imakemdep.h (include file to build imake)

The Makefile is generated using the imake utility used in the X Window System. Here are the steps you will have to follow:

1. Find a preprocessor (cpp) symbol that can be used to identify the target platform. If the preprocessor defines no special symbol, choose one. You will have to spec-

ify this symbol when executing the boot script. On systems whose cpp does not define a unique preprocessor symbol, you must add the one you have chosen to the end of the cpp_argv table in config/imakemdep.h

2. Edit the file Imake.tmpl, and add a section for your platform, for example:

   ```
   #ifdef foo
   #define MacroIncludeFile <foo.cf>
   #define MacroFile foo.cf
   #undef foo
   #define FOOArchitecture
   #endif /* foo */
   ```

   Where foo is the preprocessor symbol which uniquely identifies the platform or the symbol chosen in step one.

3. Write the configuration file foo.cf. The best way is to copy an existing .cf file, and modify it for the new platform. It is advisable to start from the configuration file for one of the reference platforms listed above. Have a look at various .cf files to find which parameters may be useful to you.

   You should read the man pages for cc and ld to choose the options to compile and to build the library (static and dynamic).

4. Run config/boot. If you found no preprocessor symbol that uniquely identifies the platform, you have to specify it at the Bootstrap flags prompt, for example: -Dfoo. You may also have to specify additional bootstrap flags that will be used when compiling imake. For instance, on most System V-based platforms, you must define the flag SYSV. So, if your platform foo is a System V, you would have to type:

   ```
   Bootstrap flags ? [] -Dfoo -DSYSV
   ```

   If imake compiles without problems, execute the commands listed above. If this fails, you should modify your configuration file and try again.

To learn more about imake, read the file config/doc/README.imake. It explains how to use imake with X11, not XFaceMaker, but the principles are the same. You can also read *An Imake Tutorial* by Mark Moraes, *Using Imake to Configure the X Window System* by Paul DuBois, or *Software Portability With IMAKE* by Paul Dubois published by O'Reilly & Associates (it is a nutshell). Please note that NSL does not support the imake utility.

The only non-standard defines used by NSL are:

- -DMOTIF_1_2 : required if your target platform has MOTIF_1_2
- -DHAS_GLS required if your target platform has General Language Support

(X/OPEN)

APPENDIX A: Functions Called in FACE

This Appendix contains a complete list of the functions attached by XFaceMaker for use in FACE. The FACE names are included for backward compatibility only. We recommend that you use the C name.

1 C Functions:

C Name	FACE Name
atoi	atoi
exit	Exit, Quit
fclose	fclose
fgets	fgets
fopen	fopen
fputs	fputs
free	free
fscanf	fscanf
malloc	malloc
pclose	pclose
popen	popen
printf	printf
sprintf	sprintf
sscanf	sscanf
strcat	strcat
strcmp	strcmp
strcpy	strcpy
strlen	strlen
strncmp	strncmp
strncpy	strncpy

2 Xt Functions:

C Name	FACE Name
XtAddGrab	XtAddGrab
XtAugmentTranslations	XtAugmentTranslations
XtCallActionProc	XtCallActionProc
XtCallCallbacks	XtCallCallbacks
XtClass	XtClass
XtDestroyWidget	XtDestroyWidget
XtDisplay	XtDisplay
XtDisplayOfObject	XtDisplayOfObject
XtFree	XtFree
XtIsForm	XtIsForm
XtIsManaged	XtIsManaged
XtIsSensitive	XtIsSensitive
XtIsSubclass	XtIsSubclass
XtMalloc	XtMalloc
XtManageChild	XtManageChild
XtMapWidget	XtMapWidget
XtMoveWidget	XtMoveWidget
XtName	XtName
XtNameToWidget	XtNameToWidget
XtNewString	XtNewString
XtOverrideTranslations	XtOverrideTranslations
XtParent	XtParent
XtPopdown	XtPopdown
XtPopup	XtPopup
XtRealizeWidget	XtRealizeWidget
XtRemoveGrab	XtRemoveGrab
XtSetKeyboardFocus	XtSetKeyboardFocus
XtSetSensitive	XtSetSensitive
XtSuperclass	XtSuperclass
XtUnmanageChild	XtUnmanageChild

C Name	FACE Name
XtUnmapWidget	XtUnmapWidget
XtUnrealizeWidget	XtUnrealizeWidget
XtWarning	XtWarning
XtWindowOfObject	XtWindow

3 FACE or FmFunctions:

C Name	FACE Name
-	breakpoint
-	cbtoi
-	cftoi
-	citof
-	clear
-	cstoi
-	data
-	destroy
-	first
-	new_array
-	new_c_array
-	new_struct
-	new_table
-	next
-	random
-	return
-	size
FaceConvertString	ConvertString
FaceEvalString	eval_string
FaceInitTimer	FaceInitTimer, init_timer
FaceSetValues	SetValues
FaceStopTimer	FaceStopTimer, stop_timer
FmAddArg	FmAddArg
FmBeep	FmBeep, Beep
FmCallCallbacksTimeOut	CallCallbacksTimeOut
FmCallEditor	CallEditor
FmCallValue	FmCallValue
FmClearArgList	FmClearArgList, ClearArgList
FmCreateDragIcon	FmCreateDragIcon
FmCreateRectangle	FmCreateRectangle

C Name	FACE Name
FmCreateXmString	CreateXmString
FmDeleteObject	FmDeleteObject
FmDisableTraversal	DisableTraversal
FmDoEvent	FmDoEvent
FmDragStart	FmDragStart
FmDropSiteRegister	FmDropSiteRegister
FmDropSiteRetrieve	FmDropSiteRetrieve
FmDropSiteUpdate	FmDropSiteUpdate
FmDropTransferAdd	FmDropTransferAdd
FmDropTransferStart	FmDropTransferStart
FmDtoi	dtoi
FmEnableTraversal	EnableTraversal
FmEqualString	FmEqualString, equal
FmFreeArgList	FmFreeArgList
FmFtoi	ftoi
FmGetActiveValue	GetActiveValue
FmGetActiveValueAddr	GetActiveValueAddr
FmGetActiveValueTimeOut	GetActiveValueTimeOut
FmGetPixmap	FmGetPixmap
FmGetString	GetString
FmGetStringFromTable	GetStringFromTable
FmGetValue	FmGetValue
FmGetVersionString	GetVersionString
FmGetXmDisplay	FmGetXmDisplay
FmGetXmScreen	FmGetXmScreen
FmHideWidget	Hide, hide, FmHideWidget
FmInternationalize	Internationalize
FmItoa	itoa
FmItod	itod
FmItof	itof
FmListAllowKeySelection	FmListAllowKeySelection
FmListGetItems	FmListGetItems

C Name	FACE Name
FmListGetNthItem	ListGetNthItem
FmListGetNthSelectedItem	ListGetNthSelectedItem
FmListGetSelectedItems	FmListGetSelectedItems
FmListSetItems	ListSetItems
FmLowerWidget	LowerWidget
FmNewArgList	FmNewArgList, NewArgList
FmNotEqualString	not_equal
FmPopupAndWait	PopupAndWait
FmRaiseWidget	RaiseWidget
FmRegisterSimpleDropSite	FmRegisterSimpleDropSite
FmReturn	Return
FmSendClick	SendClick, send_click
FmSendMessage	SendMessage
FmSetActiveValue	SetActiveValue
FmSetActiveValueTimeOut	SetActiveValueTimeOut
FmSetTallest	FmSetTallest
FmSetValue	FmSetValue
FmSetWidest	FmSetWidest
FmShowPopup	ShowPopup
FmShowWidget	Show, show
FmStartCursorDrag	FmStartCursorDrag
FmStartSimpleDrag	FmStartSimpleDrag, StartSimpleDrag
FmStopTimeOut	StopTimeOut
FmWait	wait
FmWaitForReturn	WaitForReturn
FmWarpPointer	WarpPointer
FmWidest	Widest
FmWidgetName	WidgetName
SuperclassDeleteChild	SuperclassDeleteChild
SuperclassExpose	SuperclassExpose
SuperclassInsertChild	SuperclassInsertChild
SuperclassRealize	SuperclassRealize

C Name	FACE Name
SuperclassResize	SuperclassResize

4 Motif Functions

C Name	FACE Name
XmActivateProtocol	XmActivateProtocol
XmAddProtocolCallback	XmAddProtocolCallback
XmAddProtocols	XmAddProtocols
XmAddTabGroup	XmAddTabGroup
XmCascadeButtonGadgetHighlight	XmCascadeButtonGadgetHighlight
XmCascadeButtonHighlight	XmCascadeButtonHighlight
XmChangeColor	XmChangeColor
XmClipboardBeginCopy	XmClipboardBeginCopy
XmClipboardCancelCopy	XmClipboardCancelCopy
XmClipboardCopy	XmClipboardCopy
XmClipboardCopyByName	XmClipboardCopyByName
XmClipboardEndCopy	XmClipboardEndCopy
XmClipboardEndRetrieve	XmClipboardEndRetrieve
XmClipboardInquireCount	XmClipboardInquireCount
XmClipboardInquireFormat	XmClipboardInquireFormat
XmClipboardInquireLength	XmClipboardInquireLength
XmClipboardInquirePendingItems	XmClipboardInquirePendingItems
XmClipboardLock	XmClipboardLock
XmClipboardRegisterFormat	XmClipboardRegisterFormat
XmClipboardRetrieve	XmClipboardRetrieve
XmClipboardStartCopy	XmClipboardStartCopy
XmClipboardStartRetrieve	XmClipboardStartRetrieve
XmClipboardUndoCopy	XmClipboardUndoCopy
XmClipboardUnlock	XmClipboardUnlock
XmClipboardWithdrawFormat	XmClipboardWithdrawFormat
XmCommandAppendValue	XmCommandAppendValue

C Name	FACE Name
XmCommandError	XmCommandError
XmCommandGetChild	XmCommandGetChild
XmCommandSetValue	XmCommandSetValue
XmConvertUnits	XmConvertUnits
XmCreateArrowButton	XmCreateArrowButton
XmCreateArrowButtonGadget	XmCreateArrowButtonGadget
XmCreateBulletinBoard	XmCreateBulletinBoard
XmCreateBulletinBoardDialog	XmCreateBulletinBoardDialog
XmCreateCascadeButton	XmCreateCascadeButton
XmCreateCascadeButtonGadget	XmCreateCascadeButtonGadget
XmCreateCommand	XmCreateCommand
XmCreateCommandDialog	XmCreateCommandDialog
XmCreateDialogShell	XmCreateDialogShell
XmCreateDragIcon	XmCreateDragIcon
XmCreateDrawingArea	XmCreateDrawingArea
XmCreateDrawnButton	XmCreateDrawnButton
XmCreateErrorDialog	XmCreateErrorDialog
XmCreateFileSelectionBox	XmCreateFileSelectionBox
XmCreateFileSelectionDialog	XmCreateFileSelectionDialog
XmCreateForm	XmCreateForm
XmCreateFormDialog	XmCreateFormDialog
XmCreateFrame	XmCreateFrame
XmCreateInformationDialog	XmCreateInformationDialog
XmCreateLabel	XmCreateLabel
XmCreateLabelGadget	XmCreateLabelGadget
XmCreateList	XmCreateList
XmCreateMainWindow	XmCreateMainWindow
XmCreateMenuBar	XmCreateMenuBar
XmCreateMenuShell	XmCreateMenuShell
XmCreateMessageBox	XmCreateMessageBox
XmCreateMessageDialog	XmCreateMessageDialog
XmCreateOptionMenu	XmCreateOptionMenu

C Name	FACE Name
XmCreatePanedWindow	XmCreatePanedWindow
XmCreatePopupMenu	XmCreatePopupMenu
XmCreatePromptDialog	XmCreatePromptDialog
XmCreatePulldownMenu	XmCreatePulldownMenu
XmCreatePushButton	XmCreatePushButton
XmCreatePushButtonGadget	XmCreatePushButtonGadget
XmCreateQuestionDialog	XmCreateQuestionDialog
XmCreateRadioBox	XmCreateRadioBox
XmCreateRowColumn	XmCreateRowColumn
XmCreateScale	XmCreateScale
XmCreateScrollBar	XmCreateScrollBar
XmCreateScrolledList	XmCreateScrolledList
XmCreateScrolledText	XmCreateScrolledText
XmCreateScrolledWindow	XmCreateScrolledWindow
XmCreateSelectionBox	XmCreateSelectionBox
XmCreateSelectionDialog	XmCreateSelectionDialog
XmCreateSeparator	XmCreateSeparator
XmCreateSeparatorGadget	XmCreateSeparatorGadget
XmCreateSimpleCheckBox	XmCreateSimpleCheckBox
XmCreateSimpleMenuBar	XmCreateSimpleMenuBar
XmCreateSimpleOptionMenu	XmCreateSimpleOptionMenu
XmCreateSimplePopupMenu	XmCreateSimplePopupMenu
XmCreateSimplePulldownMenu	XmCreateSimplePulldownMenu
XmCreateSimpleRadioBox	XmCreateSimpleRadioBox
XmCreateTemplateDialog	XmCreateTemplateDialog
XmCreateText	XmCreateText
XmCreateTextField	XmCreateTextField
XmCreateToggleButton	XmCreateToggleButton
XmCreateToggleButtonGadget	XmCreateToggleButtonGadget
XmCreateWarningDialog	XmCreateWarningDialog
XmCreateWorkArea	XmCreateWorkArea
XmCreateWorkingDialog	XmCreateWorkingDialog

C Name	FACE Name
XmCvtCTToXmString	XmCvtCTToXmString
XmCvtFromHorizontalPixels	XmCvtFromHorizontalPixels
XmCvtFromVerticalPixels	XmCvtFromVerticalPixels
XmCvtStringToUnitType	XmCvtStringToUnitType
XmCvtTextToXmString	XmCvtTextToXmString
XmCvtToHorizontalPixels	XmCvtToHorizontalPixels
XmCvtToVerticalPixels	XmCvtToVerticalPixels
XmCvtXmStringToCT	XmCvtXmStringToCT
XmCvtXmStringToText	XmCvtXmStringToText
XmDeactivateProtocol	XmDeactivateProtocol
XmDestroyPixmap	XmDestroyPixmap
XmDragCancel	XmDragCancel
XmDragStart	XmDragStart
XmDropSiteConfigureStackingOrder	XmDropSiteConfigureStackingOrder
XmDropSiteEndUpdate	XmDropSiteEndUpdate
XmDropSiteGetActiveVisuals	XmDropSiteGetActiveVisuals
XmDropSiteQueryStackingOrder	XmDropSiteQueryStackingOrder
XmDropSiteRegister	XmDropSiteRegister
XmDropSiteRetrieve	XmDropSiteRetrieve
XmDropSiteStartUpdate	XmDropSiteStartUpdate
XmDropSiteUnregister	XmDropSiteUnregister
XmDropSiteUpdate	XmDropSiteUpdate
XmDropTransferAdd	XmDropTransferAdd
XmDropTransferStart	XmDropTransferStart
XmFileSelectionBoxGetChild	XmFileSelectionBoxGetChild
XmFileSelectionDoSearch	XmFileSelectionDoSearch
XmFontListAdd	XmFontListAdd
XmFontListAppendEntry	XmFontListAppendEntry
XmFontListCopy	XmFontListCopy
XmFontListCreate	XmFontListCreate
XmFontListEntryCreate	XmFontListEntryCreate
XmFontListEntryFree	XmFontListEntryFree

C Name	FACE Name
XmFontListEntryGetFont	XmFontListEntryGetFont
XmFontListEntryGetTag	XmFontListEntryGetTag
XmFontListEntryLoad	XmFontListEntryLoad
XmFontListFree	XmFontListFree
XmFontListFreeFontContext	XmFontListFreeFontContext
XmFontListGetNextFont	XmFontListGetNextFont
XmFontListInitFontContext	XmFontListInitFontContext
XmFontListNextEntry	XmFontListNextEntry
XmFontListRemoveEntry	XmFontListRemoveEntry
XmGetAtomName	XmGetAtomName
XmGetColorCalculation	XmGetColorCalculation
XmGetColors	XmGetColors
XmGetDestination	XmGetDestination
XmGetDragContext	XmGetDragContext
XmGetFocusWidget	XmGetFocusWidget
XmGetMenuCursor	XmGetMenuCursor
XmGetPixmap	XmGetPixmap
XmGetPixmapByDepth	XmGetPixmapByDepth
XmGetPostedFromWidget	XmGetPostedFromWidget
XmGetSecondaryResourceData	XmGetSecondaryResourceData
XmGetTabGroup	XmGetTabGroup
XmGetTearOffControl	XmGetTearOffControl
XmGetVisibility	XmGetVisibility
XmGetXmDisplay	XmGetXmDisplay
XmGetXmScreen	XmGetXmScreen
XmImGetXIM	XmImGetXIM
XmImMbLookupString	XmImMbLookupString
XmImRegister	XmImRegister
XmImSetFocusValues	XmImSetFocusValues
XmImSetValues	XmImSetValues
XmImUnregister	XmImUnregister
XmImUnsetFocus	XmImUnsetFocus

C Name	FACE Name
XmImVaSetFocusValues	XmImVaSetFocusValues
XmImVaSetValues	XmImVaSetValues
XmInstallImage	XmInstallImage
XmInternAtom	XmInternAtom
XmIsMotifWMRunning	XmIsMotifWMRunning
XmIsTraversable	XmIsTraversable
XmListAddItem	XmListAddItem
XmListAddItemUnselected	XmListAddItemUnselected
XmListAddItems	XmListAddItems
XmListAddItemsUnselected	XmListAddItemsUnselected
XmListDeleteAllItems	XmListDeleteAllItems
XmListDeleteItem	XmListDeleteItem
XmListDeleteItems	XmListDeleteItems
XmListDeleteItemsPos	XmListDeleteItemsPos
XmListDeletePos	XmListDeletePos
XmListDeletePositions	XmListDeletePositions
XmListDeselectAllItems	XmListDeselectAllItems
XmListDeselectItem	XmListDeselectItem
XmListDeselectPos	XmListDeselectPos
XmListGetKbdItemPos	XmListGetKbdItemPos
XmListGetMatchPos	XmListGetMatchPos
XmListGetSelectedPos	XmListGetSelectedPos
XmListItemExists	XmListItemExists
XmListItemPos	XmListItemPos
XmListPosSelected	XmListPosSelected
XmListPosToBounds	XmListPosToBounds
XmListReplaceItems	XmListReplaceItems
XmListReplaceItemsPos	XmListReplaceItemsPos
XmListReplaceItemsPosUnselected	XmListReplaceItemsPosUnselected
XmListReplaceItemsUnselected	XmListReplaceItemsUnselected
XmListReplacePositions	XmListReplacePositions
XmListSelectItem	XmListSelectItem

C Name	FACE Name
XmListSelectPos	XmListSelectPos
XmListSetAddMode	XmListSetAddMode
XmListSetBottomItem	XmListSetBottomItem
XmListSetBottomPos	XmListSetBottomPos
XmListSetHorizPos	XmListSetHorizPos
XmListSetItem	XmListSetItem
XmListSetKbdItemPos	XmListSetKbdItemPos
XmListSetPos	XmListSetPos
XmListUpdateSelectedList	XmListUpdateSelectedList
XmListYToPos	XmListYToPos
XmMainWindowSep1	XmMainWindowSep1
XmMainWindowSep2	XmMainWindowSep2
XmMainWindowSep3	XmMainWindowSep3
XmMainWindowSetAreas	XmMainWindowSetAreas
XmMapSegmentEncoding	XmMapSegmentEncoding
XmMenuPosition	XmMenuPosition
XmMessageBoxGetChild	XmMessageBoxGetChild
XmOptionButtonGadget	XmOptionButtonGadget
XmOptionLabelGadget	XmOptionLabelGadget
XmProcessTraversal	XmProcessTraversal
XmRegisterConverters	XmRegisterConverters
XmRegisterSegmentEncoding	XmRegisterSegmentEncoding
XmRemoveProtocolCallback	XmRemoveProtocolCallback
XmRemoveProtocols	XmRemoveProtocols
XmRemoveTabGroup	XmRemoveTabGroup
XmRepTypeAddReverse	XmRepTypeAddReverse
XmRepTypeGetId	XmRepTypeGetId
XmRepTypeGetNameList	XmRepTypeGetNameList
XmRepTypeGetRecord	XmRepTypeGetRecord
XmRepTypeGetRegistered	XmRepTypeGetRegistered
XmRepTypeInstallTearOffModelConverter	XmRepTypeInstallTearOffModelConverter
XmRepTypeRegister	XmRepTypeRegister

C Name	FACE Name
XmRepTypeValidValue	XmRepTypeValidValue
XmResolveAllPartOffsets	XmResolveAllPartOffsets
XmResolvePartOffsets	XmResolvePartOffsets
XmScaleGetValue	XmScaleGetValue
XmScaleSetValue	XmScaleSetValue
XmScrollBarGetValues	XmScrollBarGetValues
XmScrollBarSetValues	XmScrollBarSetValues
XmScrollVisible	XmScrollVisible
XmScrolledWindowSetAreas	XmScrolledWindowSetAreas
XmSelectionBoxGetChild	XmSelectionBoxGetChild
XmSetColorCalculation	XmSetColorCalculation
XmSetFontUnit	XmSetFontUnit
XmSetFontUnits	XmSetFontUnits
XmSetMenuCursor	XmSetMenuCursor
XmSetProtocolHooks	XmSetProtocolHooks
XmStringBaseline	XmStringBaseline
XmStringByteCompare	XmStringByteCompare
XmStringCompare	XmStringCompare
XmStringConcat	XmStringConcat
XmStringCopy	XmStringCopy
XmStringCreate	XmStringCreate
XmStringCreateFontList	XmStringCreateFontList
XmStringCreateLocalized	XmStringCreateLocalized
XmStringCreateLtoR	XmStringCreateLtoR
XmStringCreateSimple	XmStringCreateSimple
XmStringDirectionCreate	XmStringDirectionCreate
XmStringEmpty	XmStringEmpty
XmStringExtent	XmStringExtent
XmStringFree	XmStringFree
XmStringFreeContext	XmStringFreeContext
XmStringGetLtoR	XmStringGetLtoR
XmStringGetNextComponent	XmStringGetNextComponent

C Name	FACE Name
XmStringGetNextSegment	XmStringGetNextSegment
XmStringHasSubstring	XmStringHasSubstring
XmStringHeight	XmStringHeight
XmStringInitContext	XmStringInitContext
XmStringLength	XmStringLength
XmStringLineCount	XmStringLineCount
XmStringLtoRCreate	XmStringLtoRCreate
XmStringNConcat	XmStringNConcat
XmStringNCopy	XmStringNCopy
XmStringPeekNextComponent	XmStringPeekNextComponent
XmStringSegmentCreate	XmStringSegmentCreate
XmStringSeparatorCreate	XmStringSeparatorCreate
XmStringWidth	XmStringWidth
XmTargetsAreCompatible	XmTargetsAreCompatible
XmTextClearSelection	XmTextClearSelection
XmTextCopy	XmTextCopy
XmTextCut	XmTextCut
XmTextDisableRedisplay	XmTextDisableRedisplay
XmTextEnableRedisplay	XmTextEnableRedisplay
XmTextFieldClearSelection	XmTextFieldClearSelection
XmTextFieldCopy	XmTextFieldCopy
XmTextFieldCut	XmTextFieldCut
XmTextFieldGetAddMode	XmTextFieldGetAddMode
XmTextFieldGetBaseline	XmTextFieldGetBaseline
XmTextFieldGetCursorPosition	XmTextFieldGetCursorPosition
XmTextFieldGetEditable	XmTextFieldGetEditable
XmTextFieldGetInsertionPosition	XmTextFieldGetInsertionPosition
XmTextFieldGetLastPosition	XmTextFieldGetLastPosition
XmTextFieldGetMaxLength	XmTextFieldGetMaxLength
XmTextFieldGetSelection	XmTextFieldGetSelection
XmTextFieldGetSelectionPosition	XmTextFieldGetSelectionPosition
XmTextFieldGetSelectionWcs	XmTextFieldGetSelectionWcs

C Name	FACE Name
XmTextFieldGetString	XmTextFieldGetString
XmTextFieldGetStringWcs	XmTextFieldGetStringWcs
XmTextFieldGetSubstring	XmTextFieldGetSubstring
XmTextFieldGetSubstringWcs	XmTextFieldGetSubstringWcs
XmTextFieldInsert	XmTextFieldInsert
XmTextFieldInsertWcs	XmTextFieldInsertWcs
XmTextFieldPaste	XmTextFieldPaste
XmTextFieldPosToXY	XmTextFieldPosToXY
XmTextFieldRemove	XmTextFieldRemove
XmTextFieldReplace	XmTextFieldReplace
XmTextFieldReplaceWcs	XmTextFieldReplaceWcs
XmTextFieldSetAddMode	XmTextFieldSetAddMode
XmTextFieldSetCursorPosition	XmTextFieldSetCursorPosition
XmTextFieldSetEditable	XmTextFieldSetEditable
XmTextFieldSetHighlight	XmTextFieldSetHighlight
XmTextFieldSetInsertionPosition	XmTextFieldSetInsertionPosition
XmTextFieldSetMaxLength	XmTextFieldSetMaxLength
XmTextFieldSetSelection	XmTextFieldSetSelection
XmTextFieldSetString	XmTextFieldSetString
XmTextFieldSetStringWcs	XmTextFieldSetStringWcs
XmTextFieldShowPosition	XmTextFieldShowPosition
XmTextFieldXYToPos	XmTextFieldXYToPos
XmTextFindString	XmTextFindString
XmTextFindStringWcs	XmTextFindStringWcs
XmTextGetAddMode	XmTextGetAddMode
XmTextGetBaseline	XmTextGetBaseline
XmTextGetCursorPosition	XmTextGetCursorPosition
XmTextGetEditable	XmTextGetEditable
XmTextGetInsertionPosition	XmTextGetInsertionPosition
XmTextGetLastPosition	XmTextGetLastPosition
XmTextGetMaxLength	XmTextGetMaxLength
XmTextGetSelection	XmTextGetSelection

C Name	FACE Name
XmTextGetSelectionPosition	XmTextGetSelectionPosition
XmTextGetSelectionWcs	XmTextGetSelectionWcs
XmTextGetSource	XmTextGetSource
XmTextGetString	XmTextGetString
XmTextGetStringWcs	XmTextGetStringWcs
XmTextGetSubstring	XmTextGetSubstring
XmTextGetSubstringWcs	XmTextGetSubstringWcs
XmTextGetTopCharacter	XmTextGetTopCharacter
XmTextInsert	XmTextInsert
XmTextInsertWcs	XmTextInsertWcs
XmTextPaste	XmTextPaste
XmTextPosToXY	XmTextPosToXY
XmTextRemove	XmTextRemove
XmTextReplace	XmTextReplace
XmTextReplaceWcs	XmTextReplaceWcs
XmTextScroll	XmTextScroll
XmTextSetAddMode	XmTextSetAddMode
XmTextSetCursorPosition	XmTextSetCursorPosition
XmTextSetEditable	XmTextSetEditable
XmTextSetHighlight	XmTextSetHighlight
XmTextSetInsertionPosition	XmTextSetInsertionPosition
XmTextSetMaxLength	XmTextSetMaxLength
XmTextSetSelection	XmTextSetSelection
XmTextSetSource	XmTextSetSource
XmTextSetString	XmTextSetString
XmTextSetStringWcs	XmTextSetStringWcs
XmTextSetTopCharacter	XmTextSetTopCharacter
XmTextShowPosition	XmTextShowPosition
XmTextXYToPos	XmTextXYToPos
XmToggleButtonGadgetGetState	XmToggleButtonGadgetGetState
XmToggleButtonGadgetSetState	XmToggleButtonGadgetSetState
XmToggleButtonGetState	XmToggleButtonGetState

C Name	FACE Name
XmToggleButtonSetState	XmToggleButtonSetState
XmTrackingEvent	XmTrackingEvent
XmTrackingLocate	XmTrackingLocate
XmTranslateKey	XmTranslateKey
XmUninstallImage	XmUninstallImage
XmUpdateDisplay	XmUpdateDisplay
XmVaCreateSimpleCheckBox	XmVaCreateSimpleCheckBox
XmVaCreateSimpleMenuBar	XmVaCreateSimpleMenuBar
XmVaCreateSimpleOptionMenu	XmVaCreateSimpleOptionMenu
XmVaCreateSimplePopupMenu	XmVaCreateSimplePopupMenu
XmVaCreateSimplePulldownMenu	XmVaCreateSimplePulldownMenu
XmVaCreateSimpleRadioBox	XmVaCreateSimpleRadioBox
XmWidgetGetBaselines	XmWidgetGetBaselines
XmWidgetGetDisplayRect	XmWidgetGetDisplayRect

Index